組合語言與微處理機

Yun-Nan Chang (張雲南)

Dept. of Computer Science and Engineering
National Sun Yat-Sen University

Tel: (07)5252000 ext. 4332
E-mail: ynchang@cse.nsysu.edu.tw
Office: 電資F5006

Textbook

- ARM System-on-Chip Architecture, 2nd Edition

  S. Furber

  Addison Wesley Longman, 2000

  ISBN：0-201-67519-6

  東華/新月

Reference

- ARM System Developer's Guide, Designing and Optimizing System Software
  - A N. Sloss, D Symes, C. Wright, 2004,
  - 新月圖書
- ARM assembly language, Fundamental and Techniques
  - William Hohl and Christopher Hinds, 2015,
  - 新月圖書

ARM System-on-Chip Architecture, 2nd Edition

- ***Chap. 1. An Introduction to Processor Design***
- ***Chap. 2. The ARM Architecture***
- ***Chap. 3. ARM Assembly Language Programming***
- ***Chap. 4. ARM Organization and Implementation***
- ***Chap. 5. The ARM Instruction Set***
- ***Chap. 6. Architectural Support for High-Level Languages***
- ***Chap. 7. The Thumb Instruction Set***
- ***Chap. 8. Architectural Support for System Development***
- ***Chap. 9. ARM Processor Cores***
- *Chap. 10. Memory Hierarchy*
- *Chap. 11. Architectural Support for Operating Systems*

Grades

- Midterm Exam (30%)
- Final Exam (30%)
- Homework / Quiz / Team Report(40%)
  - No written homework. Only coding assignments.
  - Group Report：原則4人/組
    - 預計10月中旬過後開始報告
    - Anything (tutorial, researches) related to ARM, assembly, embedded system...
    - 發展平台介紹：Arduino, Rasberry, PYNQ, …..

Web Site

- Course homepage: NSYSU Cyber University
  - http://cu.nsysu.edu.tw
- ARM information center (http://infocenter.arm.com)

Recent Term projects

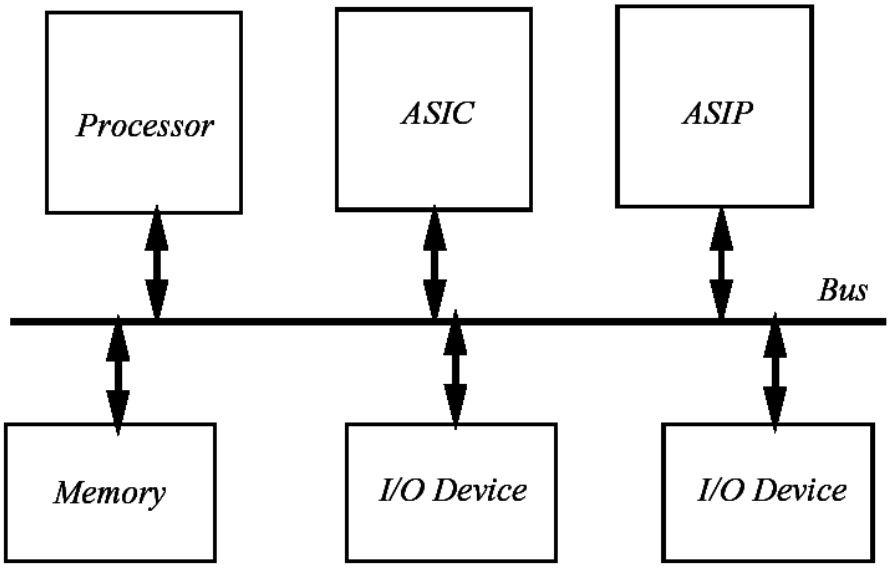
- Arduino
- Raspberry Pi
- RISC-V
- x86 ISA + example
- bootloader/start up code
- Boot sequence
- python interpreter
- PCI-e bus protocol
- ARM MMU
- DLL
- File system
- ARM tracer/debugger
- Linker
- Virtual Machine
- Linux device driver

Why study this course?

- **The bridge of**
  - **High-level (C) Programming**
  - **Computer Organization**

  => Foundation for Compiler course

- **Extend CPU to System Architecture/Applications**



各種嵌入式系統裝置

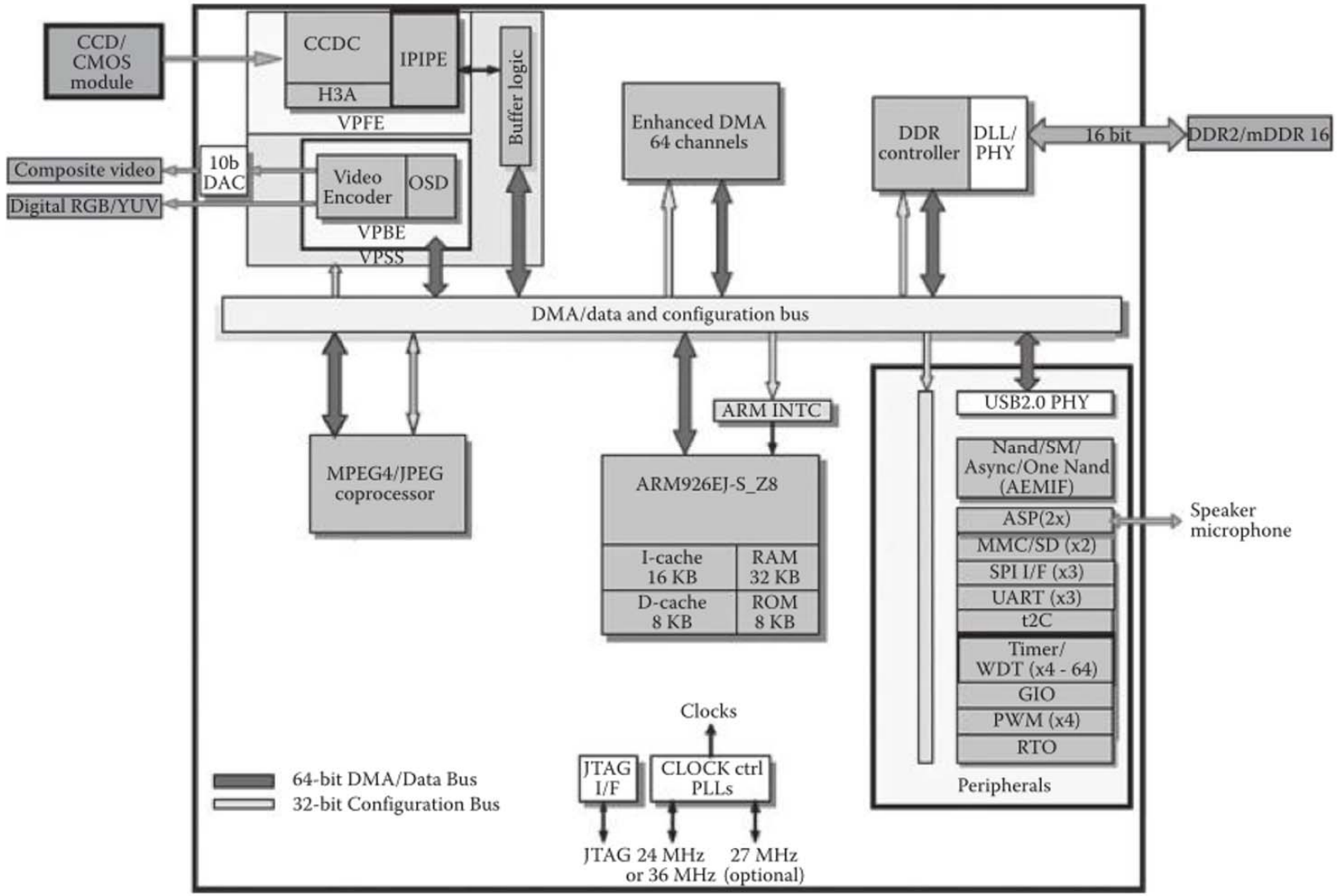
Hy-wire未來汽車

NASA火星漫遊者

Roomba機器人真空吸塵器

狗語翻譯機

The TMS320DM355 System-on-chip from Texas Instruments

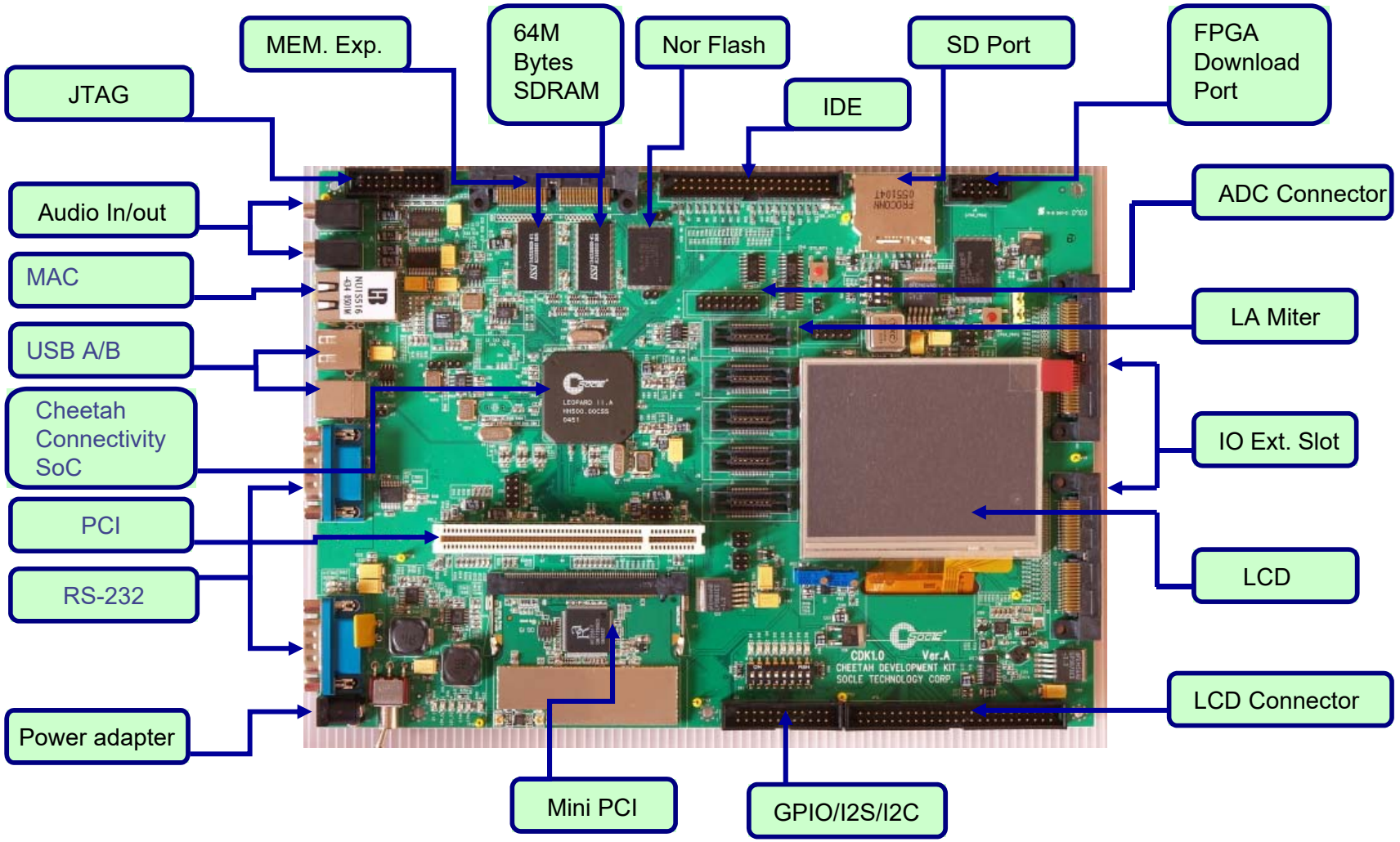


Embedded System

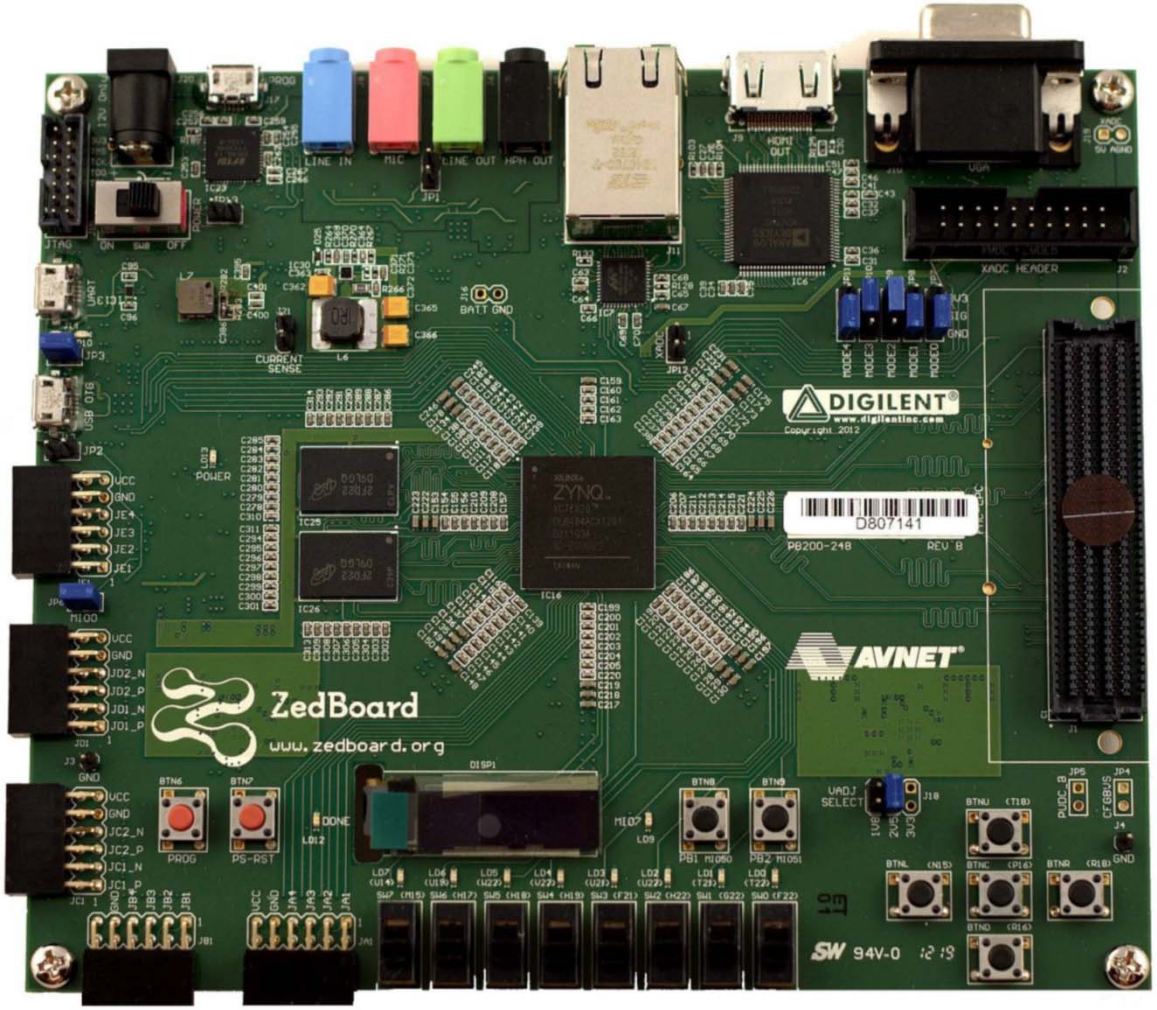
- Embedded System: a combination of computer hardware and software, and perhaps additional mechanical parts, designed to perform a specific function.
  - Microwave oven, MP3 player, intelligent appliance, …
- System components include
  - Microprocessor: execute the programming software.
  - Memory
  - Input/ouput, peripheral modules
  - Hardware modules (Application Specific Integrated Circuit)
- A simple example:

```
main(void)
{
        while (1)
        {
                toggleLed (LED_GREEN);
                delay(500);
        }
}
```

SOC board: Cheetah Development Kit (CDK)



Zedboard


ZedBoard
www.zedboard.org
DIGILENT
AVNET
ZYNQ
LINE IN
MIC
LINE OUT
HPH OUT
HDMI OUT
VGA
XADC HEADER
JTAG
POWER
PROG
PS-RST
DONE
DISP1
D807141
PB200-248
REV B

Arduino


MADE
IN ITALY
DIGITAL (PWM~)
UNO
ARDUINO
TX
RX
ON
SMD
EDITION
POWER
ANALOG IN

Chap. 1

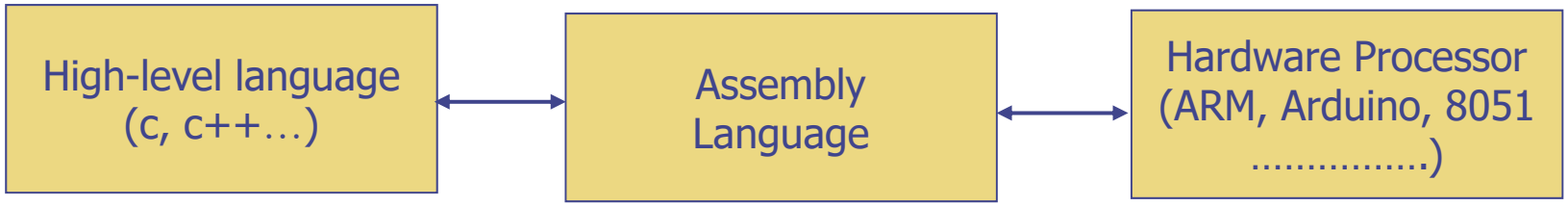
An Introduction to Processor Design

1.1 Processor architecture and organization

- Computer architecture
  - Computer **architecture** describes the user's view of the computer
  - The instruction set, visible registers, memory management table structures are all part of architecture
- Computer organization
  - Computer **organization** describes the user-invisible implementation of the architecture
  - The pipeline structure, transparent cache, table-walking hardware are all aspects of the organization

Goal

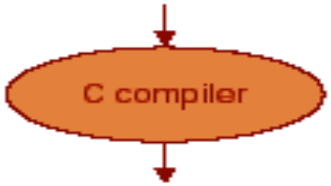
- Assembly language: A bunch of the readable, fundamental instructions that could be recognized by the processors.
  - What are these instructions ?
  - How to realize the circuits that can execute these instructions?
  - How to code the application programs using these instructions efficiently?



- Microprocessor based embedded system
  - What does the system architecture look like?
  - How to develop the applications on the system?

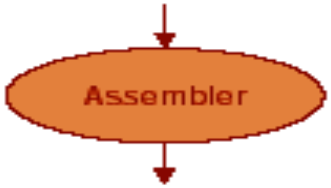
High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```



Assembly language program (for MIPS)

```
swap:
        muli $2, $5,4
        add  $2, $4,$2
        lw   $15, 0($2)
        lw   $16, 4($2)
        sw   $16, 0($2)
        sw   $15, 4($2)
        jr   $31
```



Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000100011100001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111100000000000000001000
```

1.1 Processor architecture and organization

- What is a processor
  - A general-purpose processor is a **finite-state automaton** that executes instructions held in a memory
  - The state of the system is defined by the values held in the memory locations together with the values held in certain registers within the processor itself

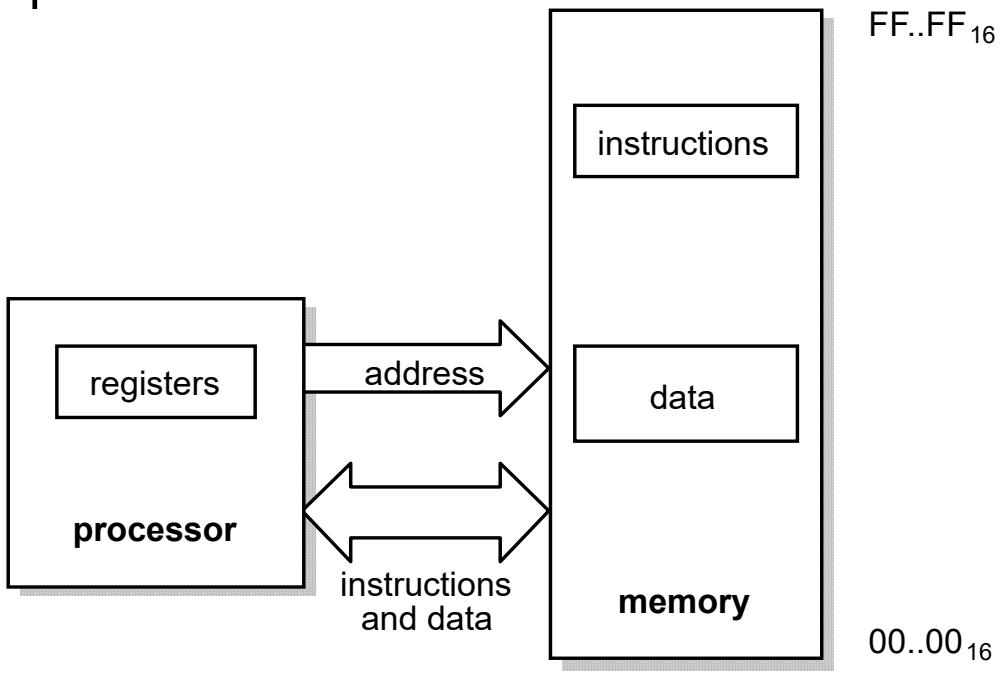
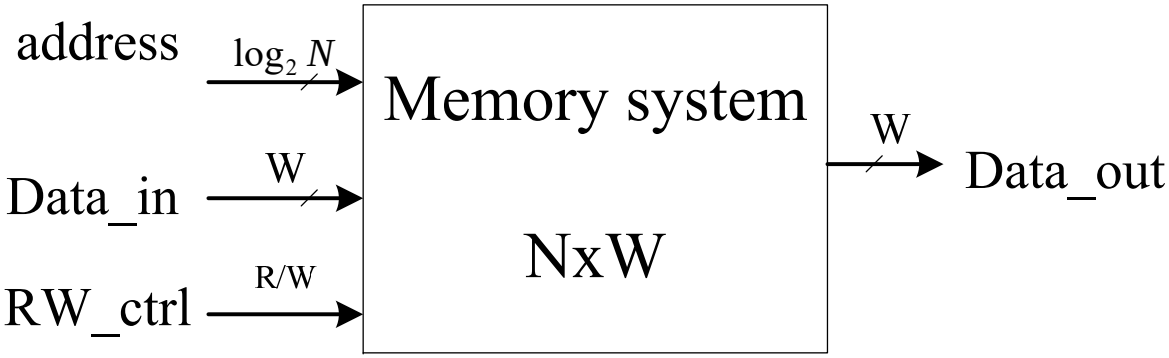
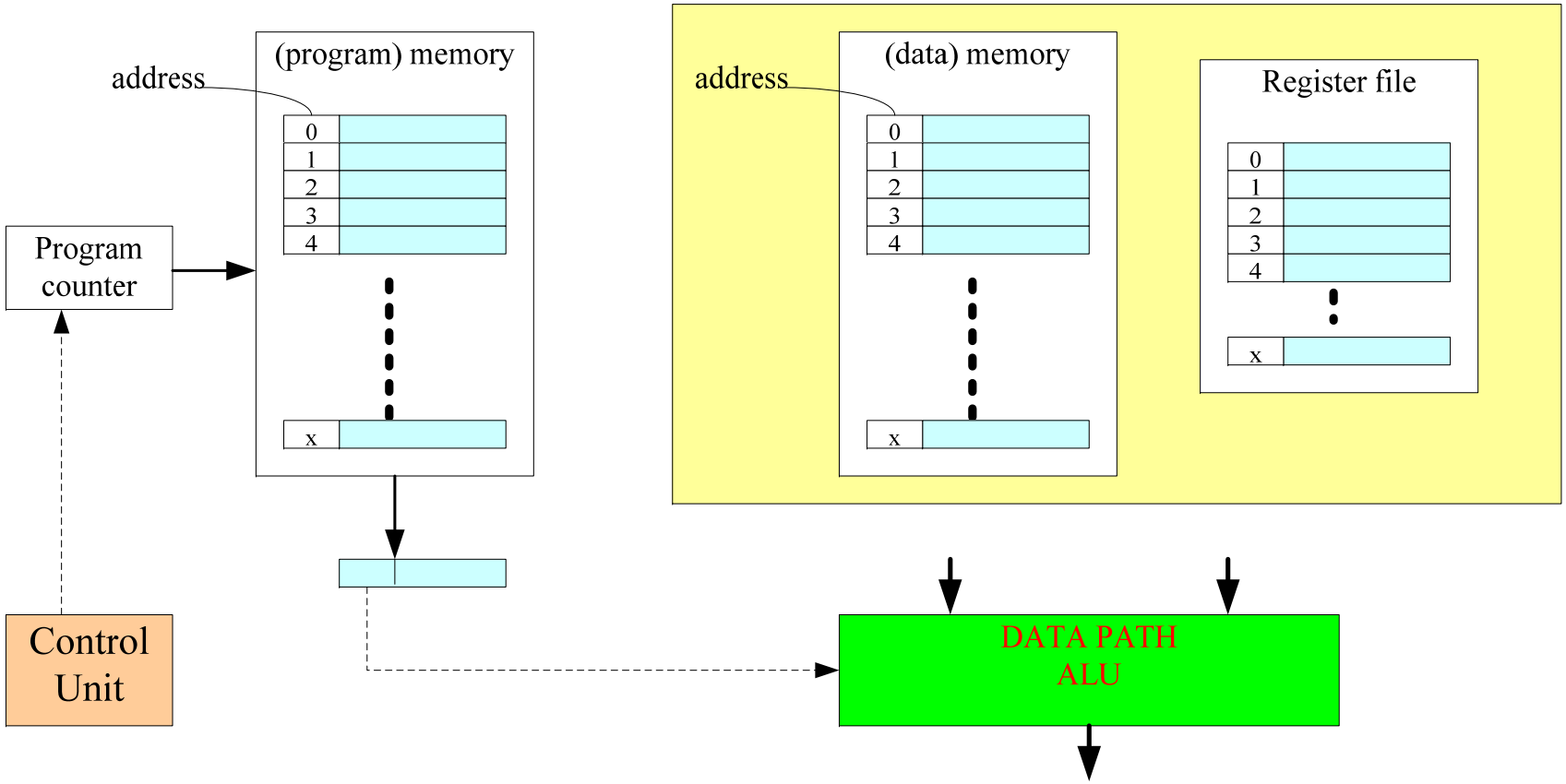
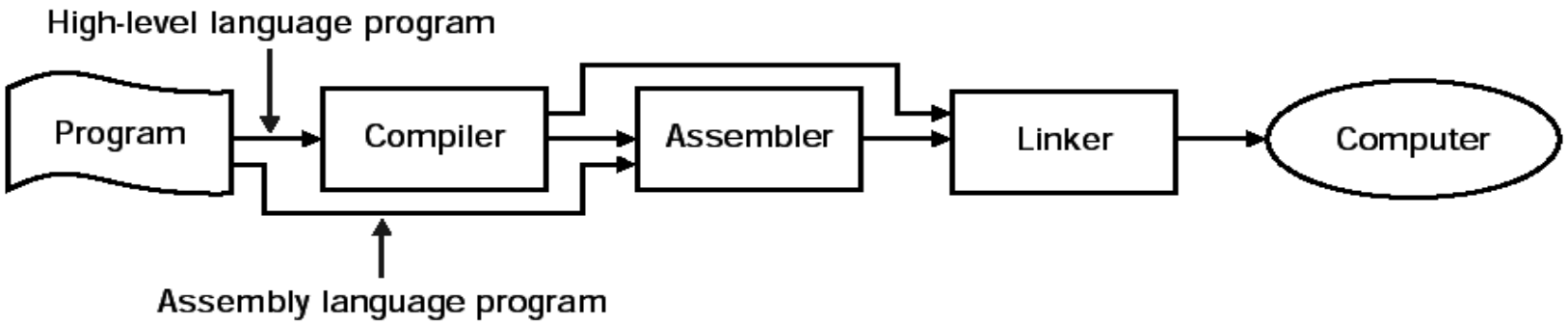


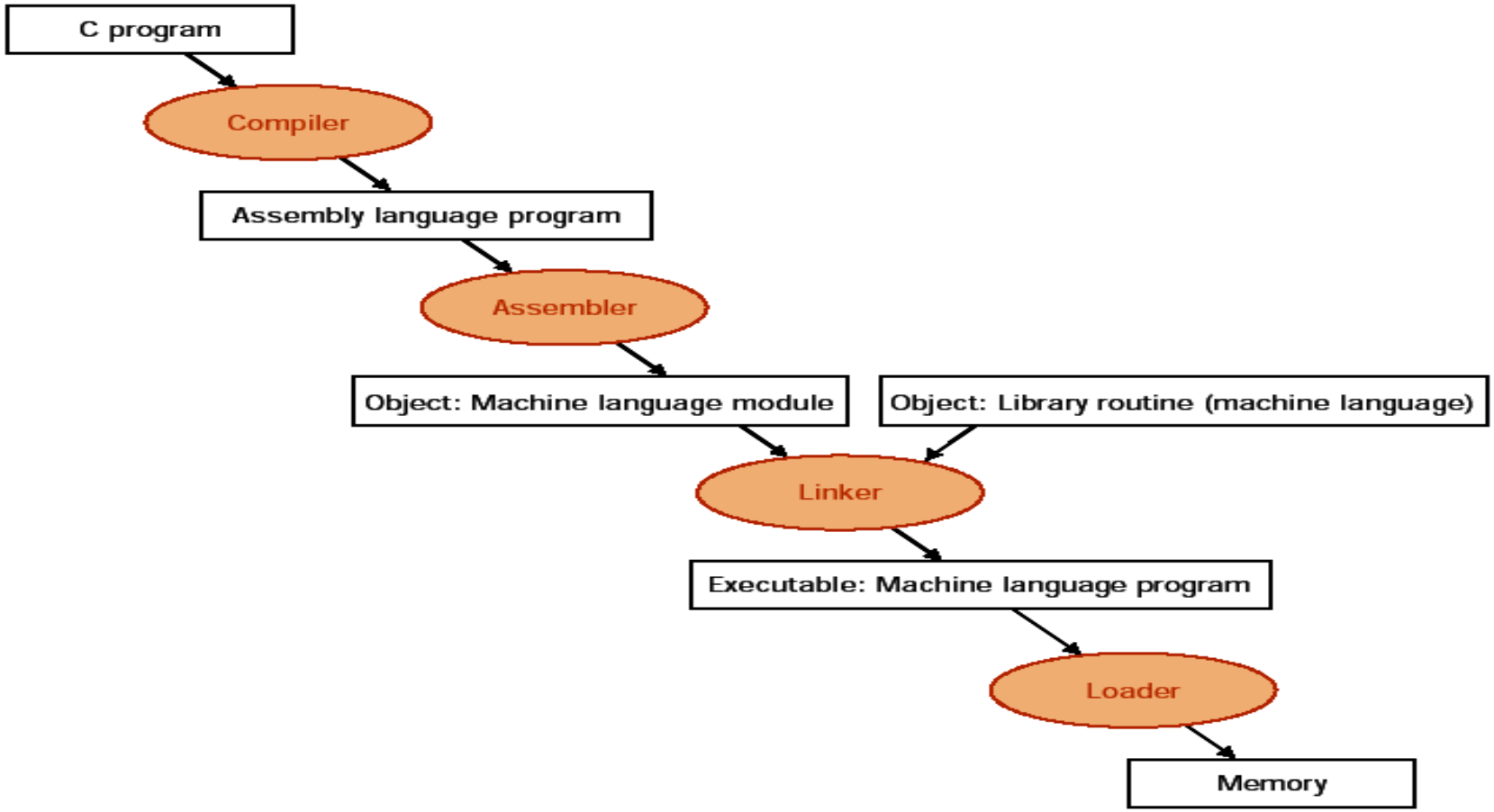
Fig. 1 The state in a stored-program digital computer

Memory block




address
(program) memory
0
1
2
3
4
x
Program counter
Control Unit
address
(data) memory
0
1
2
3
4
x
Register file
0
1
2
3
4
x
DATA PATH
ALU


High-level language program
Program
Compiler
Assembler
Linker
Computer
Assembly language program

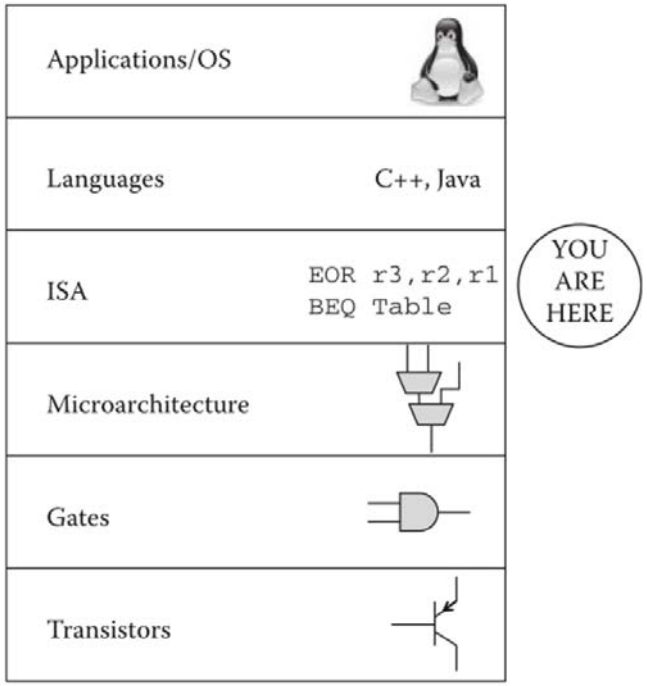

C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

1.1 Processor architecture and organization

- The stored-program computer
  - The stored-program digital computer keeps its instructions and data in the **same memory system**, allowing the instructions to be treated as data when necessary
- Computer applications
  - Because of its programmability a stored-program digital computer is universal, which means that it can undertake any task that can be described by a suitable algorithm

1.2 Abstraction in hardware design

- A modern microprocessor may be built from several million transistors each of which can switch a hundred million times a second
- A single error amongst those transitions is likely to cause the machine to collapse into a useless state
- Transistors ⇒ Logic gates ⇒ Logic symbol ⇒ True table ⇒ The gate abstraction ⇒ Levels of abstraction ⇒ Gate-level design



1.2 Abstraction in hardware design

- Levels of abstraction - A typical hierarchy of abstraction at the hardware level is:
  - Transistors
  - Logic gate, memory cells, special circuits
  - Single-bit adders, multiplexors, decoders, flip-flops
  - Word-wide adders, multiplexors, decoders, registers, buses,
  - ALUs, barrel shifters, register banks, memory blocks.
  - Processor, cache and memory management organizations
  - Processors, peripheral cells, cache memories, memory management units.
  - Integrated system chips
  - Printed circuit boards;
  - Mobile telephones, PCs, engine controllers.

1.3 MU0 – a simple processor

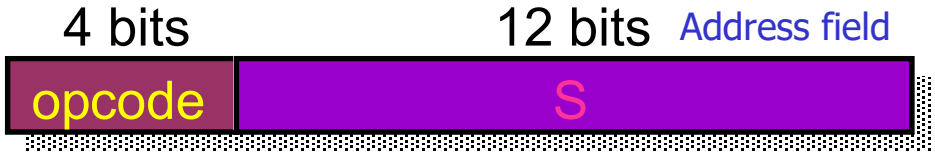
- A simple form of processor can be built from the following basic components:
  - **A program counter (PC) register**
  - **A accumulator (ACC) register**
  - **An arithmetic-logic unit (ALU)**
  - **An instruction register (IR)**
  - **Instruction decode and control logic**
- MU0 is a 16-bit machine with 12-bit address space.
  - It can address up to 8 Kbytes of memory arranged as 4096 individually addressable 16-bit location.

1.3 MU0 – a simple processor

- The MU0 **instruction set**

Instruction	Opcode	Effect
LDA S	0000	ACC := mem_{16}[S]
STO S	0001	mem_{16}[S] := ACC
ADD S	0010	ACC := ACC + mem_{16}[S]
SUB S	0011	ACC := ACC - mem_{16}[S]
JMP S	0100	PC := S
JGE S	0101	if ACC >= 0 PC := S
JNE S	0110	if ACC !=0 PC := S
STP	0111	stop

- The MU0 **instruction format**

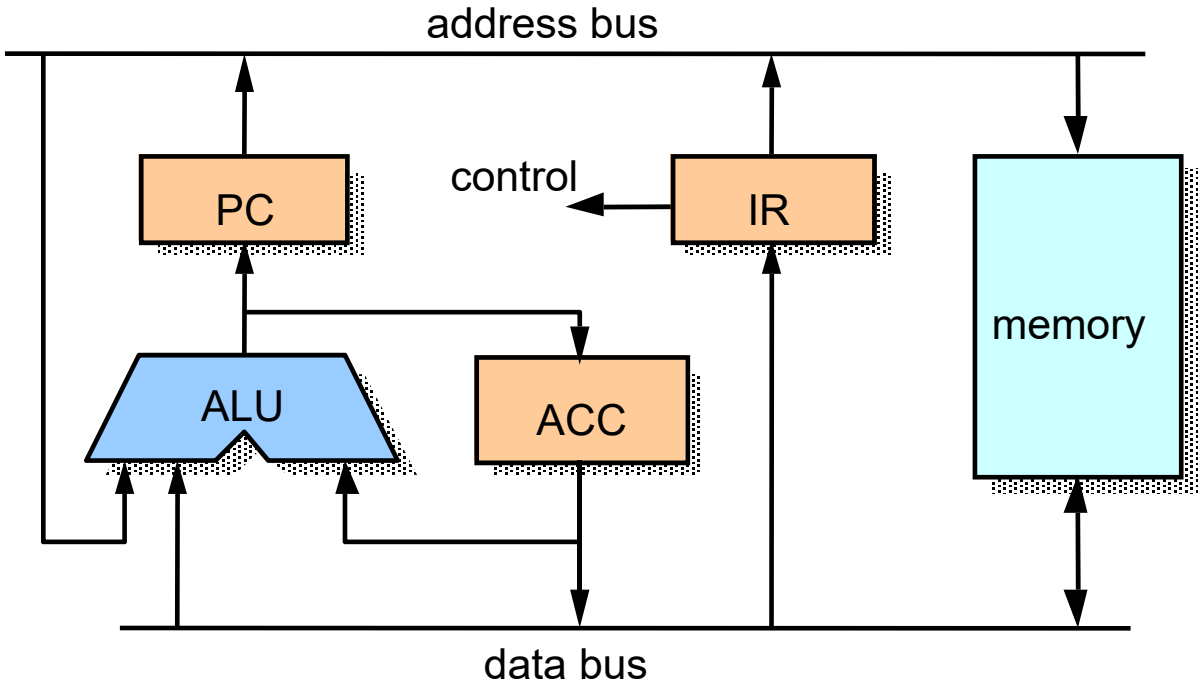


Reminding

- Memory??
- PC
- 16-bit memory, 12-bit address space
- Opcode
- Operand
  - Addressing mode
    - EX: 0010+000000001010
    - ACC+MEM[10] or ACC+#10
- Try to make connection between Assembly instruction with C instruction

1.3 MU0 – a simple processor

- MU0 **datapath** example
  - The number of clock cycles taken by the execution of each instruction depends on the number of memory accesses required.

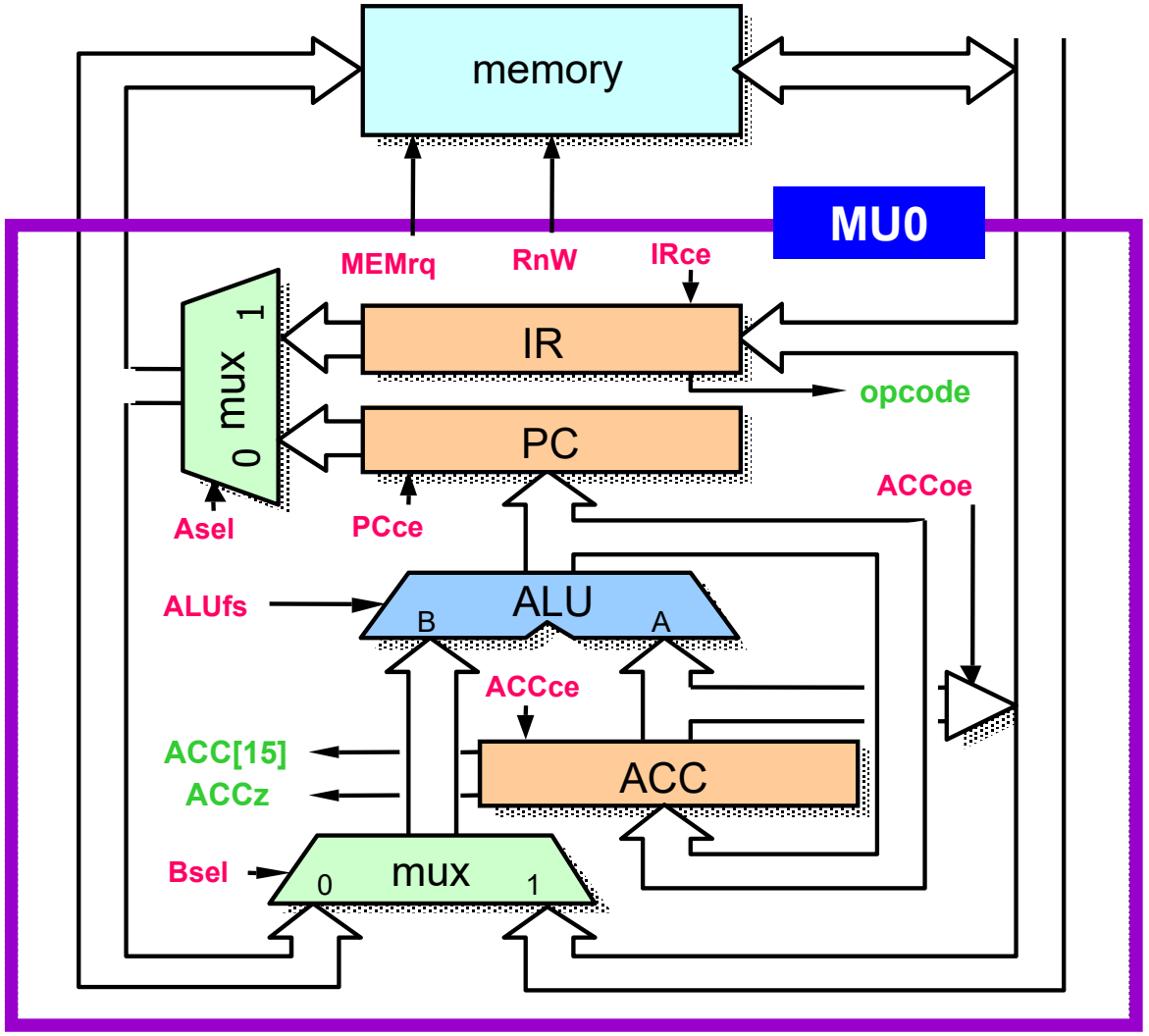


1.3 MU0 – a simple processor

- Data path operation
  - Each instruction starts when it has arrived in the instruction register.
  - Two stages of instruction execution
    - Access the memory operand and perform the desired operation.
    - Fetch the next instruction to be executed.
- Initialization
  - A reset input has to be used to bring the machine to start executing the instructions from a known address.

1.3 MU0 – a simple processor

- MU0 **register transfer level organization**



Two states
fetch,
execute

ALU functions
A+B, A-B, B,
B+1,0

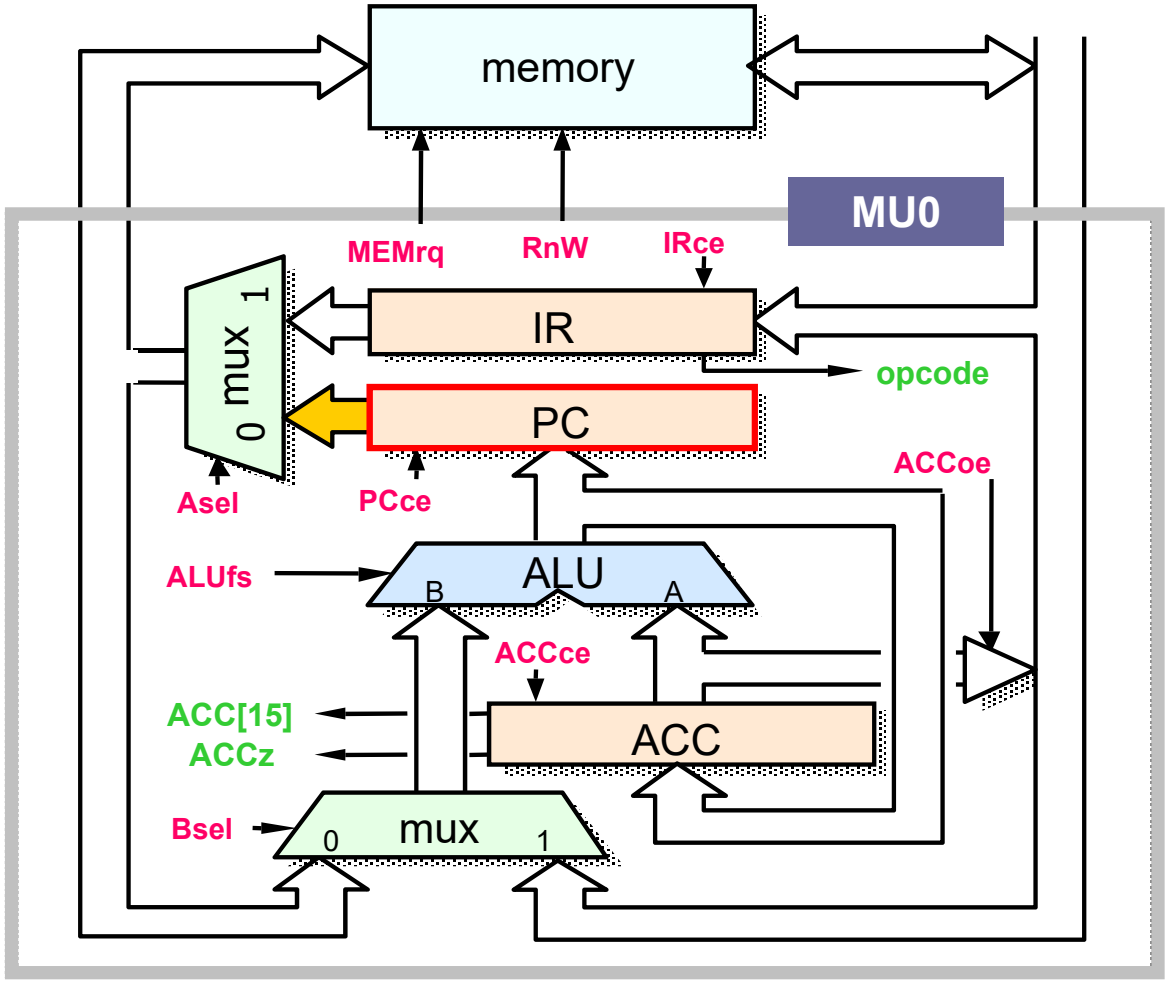
1.3 MU0 – a simple processor

- MU0 **control logic** (two states: fetch and execute)

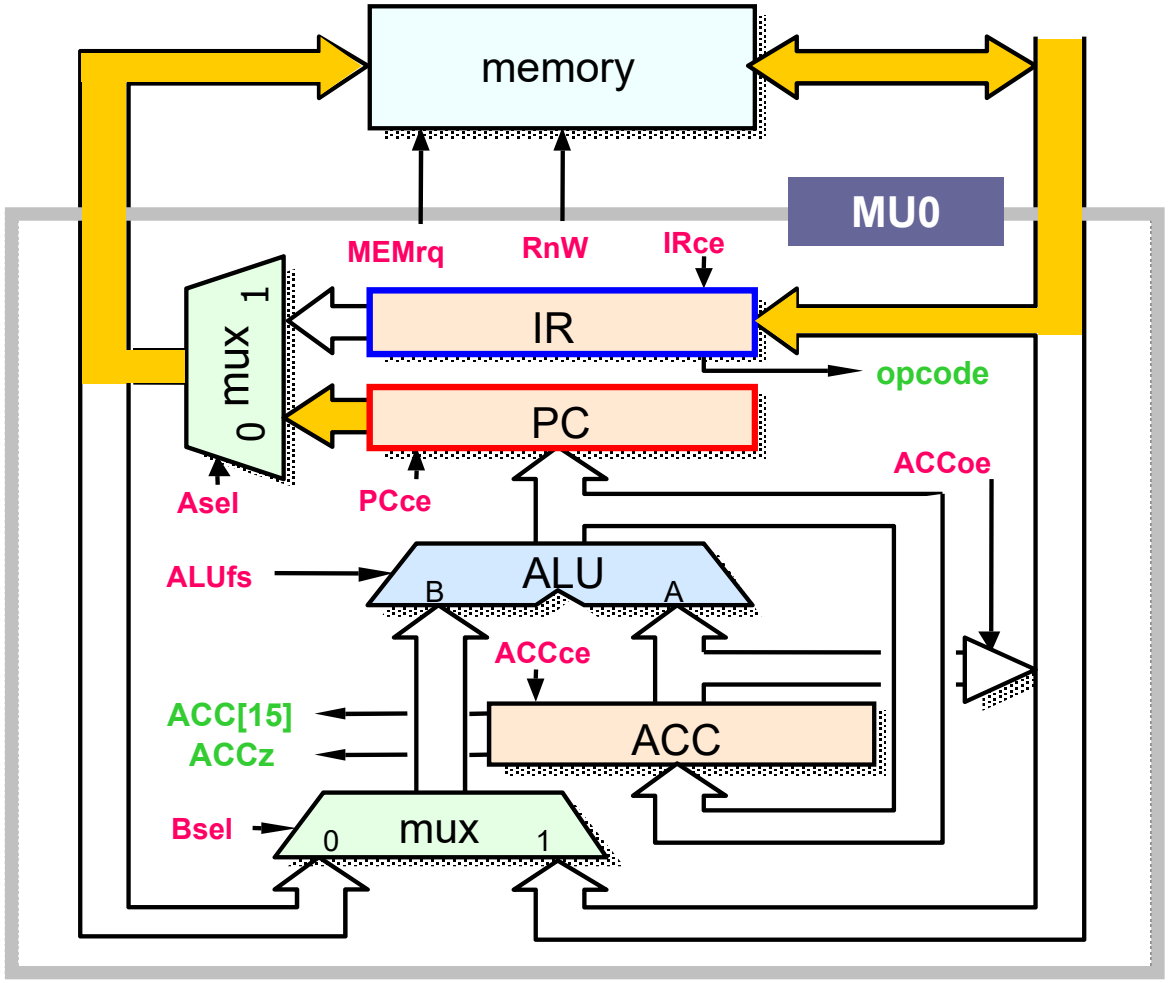
1 **0**

Inputs						Outputs									
	Opcode	Ex/ft	ACC15			Bsel		PCce		ACCoe		MEMrq	Ex/ft		
Instruction		Reset		ACCz		Asel	ACCce		IRce		ALUfs			RnW	
Reset	xxxx	1	x	x	x	0	0	1	1	1	0	=0	1	1	0 Ex
LDA S	0000	0	0	x	x	1	1	1	0	0	0	=B	1	1	1
	0000	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
STO S	0001	0	0	x	x	1	x	0	0	0	1	x	1	0 W	1
	0001	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
ADD S	0010	0	0	x	x	1	1	1	0	0	0	A+B	1	1	1
	0010	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
SUB S	0011	0	0	x	x	1	1	1	0	0	0	A-B	1	1	1
	0011	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
JMP S	0100	0	x	x	x	1	0	0	1	1	0	B+1	1	1	0
JGE S	0101	0	x	x	≥ 0	1	0	0	1	1	0	B+1	1	1	0
	0101	0	x	x	1	0	0	0	1	1	0	B+1	1	1	0
JNE S	0110	0	x	≠ 0	x	1	0	0	1	1	0	B+1	1	1	0
	0110	0	x	1	x	0	0	0	1	1	0	B+1	1	1	0
STOP	0111	0	x	x	x	1	x	0	0	0	0	x	0	1	0

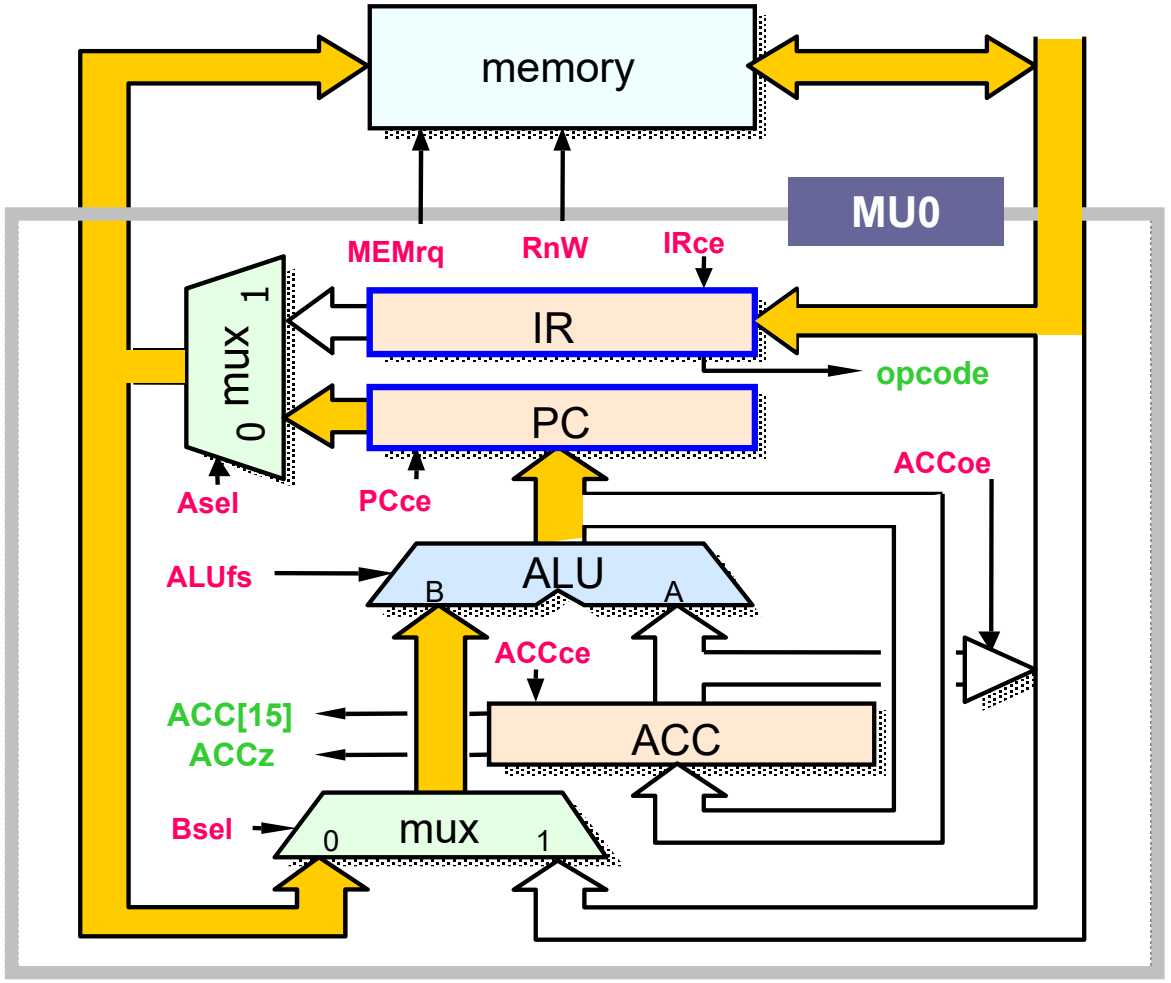
Inputs					Outputs										
Instruction	Opcode	Reset	Ex/ft	ACC15 ACCz	Asel	Bsel ACCce	PCce IRce	ACCoe	ALUfs	MEMrq RnW	Ex/ft				
Reset	xxxx	1	x	x x	0	0	1 0	1	1	0	= 0 B+1	1	1 R	0 Ex	



Inputs						Outputs										
Instruction	Opcode	Reset	Ex/ft	ACC15	ACCz	Asel	Bsel	ACCce	PCce	IRce	ACCoe	ALUfs	MEMrq	RnW	Ex/ft	
Reset	xxxx	1	x	x	x	0	0	1 0	1	1	0	=0 B+1	1	1 R	0 Ex	

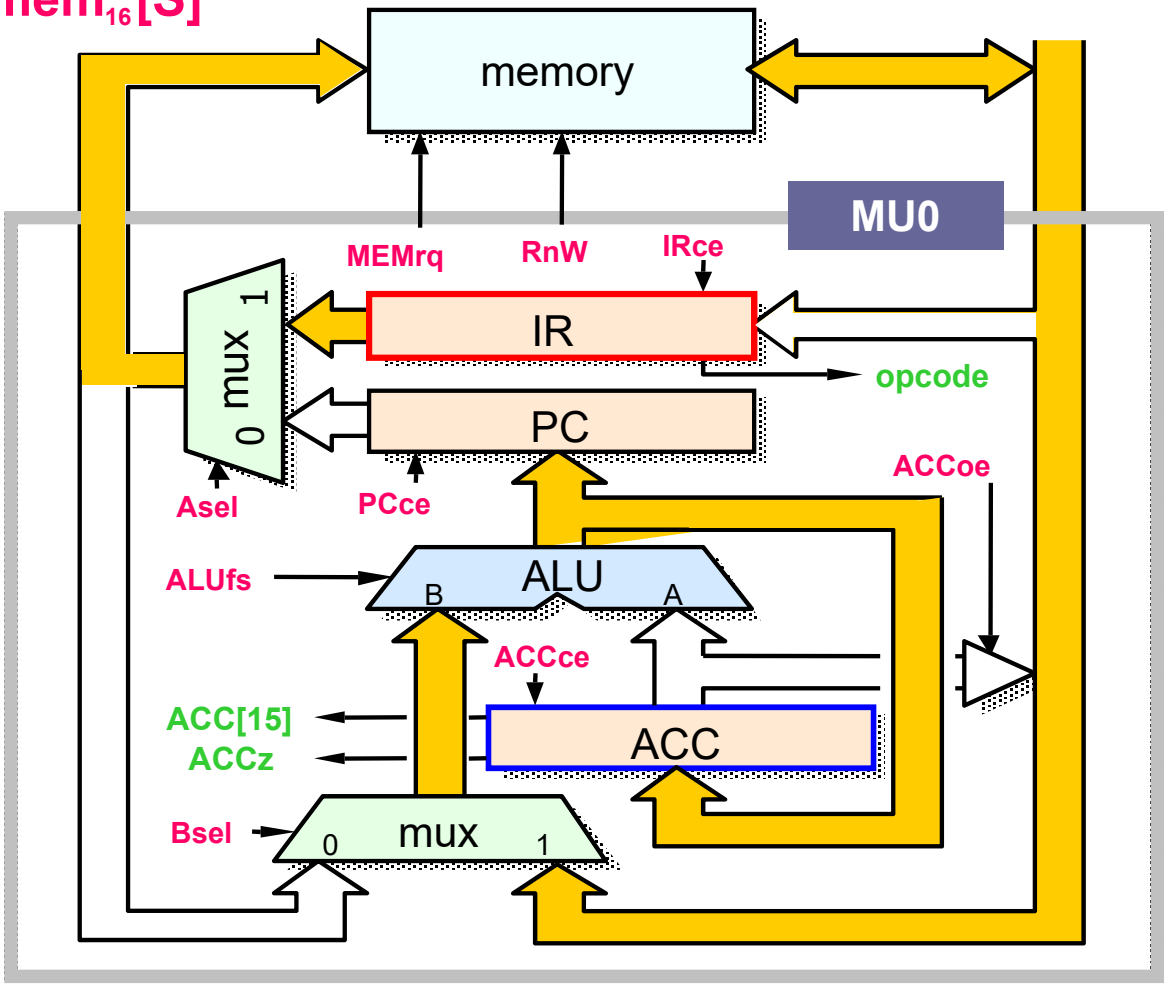


Inputs						Outputs										
Instruction	Opcode	Reset	Ex/ft	ACC15	ACCz	Asel	Bsel	ACCce	PCce	IRce	ACCoe	ALUfs	MEMrq	RnW	Ex/ft	
Reset	xxxx	1	x	x	x	0	0	1 0	1	1	0	=0 B+1	1	1 R	0 Ex	

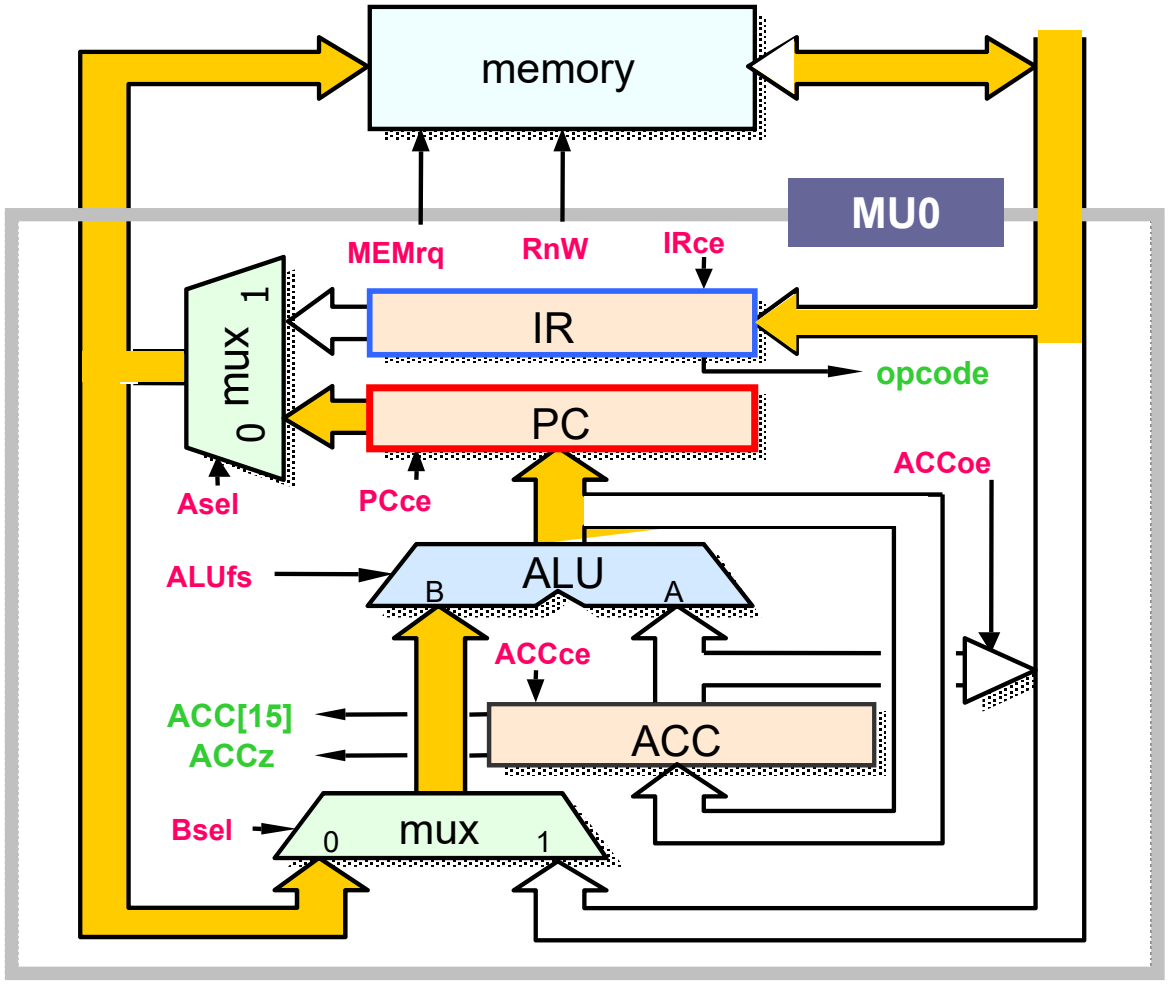


Inputs						Outputs									
Instruction	Opcode	Reset	Ex/ft	ACC15	ACCz	Asel	Bsel	ACCce	PCce	IRce	ACCoe	ALUfs	MEMrq	RnW	Ex/ft
LDA S	0000	0	0Ex	x	x	1	1	1	0	0	0	=B	1	1R	1ft
	0000	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0

LDA S ACC := mem_{16}[S]

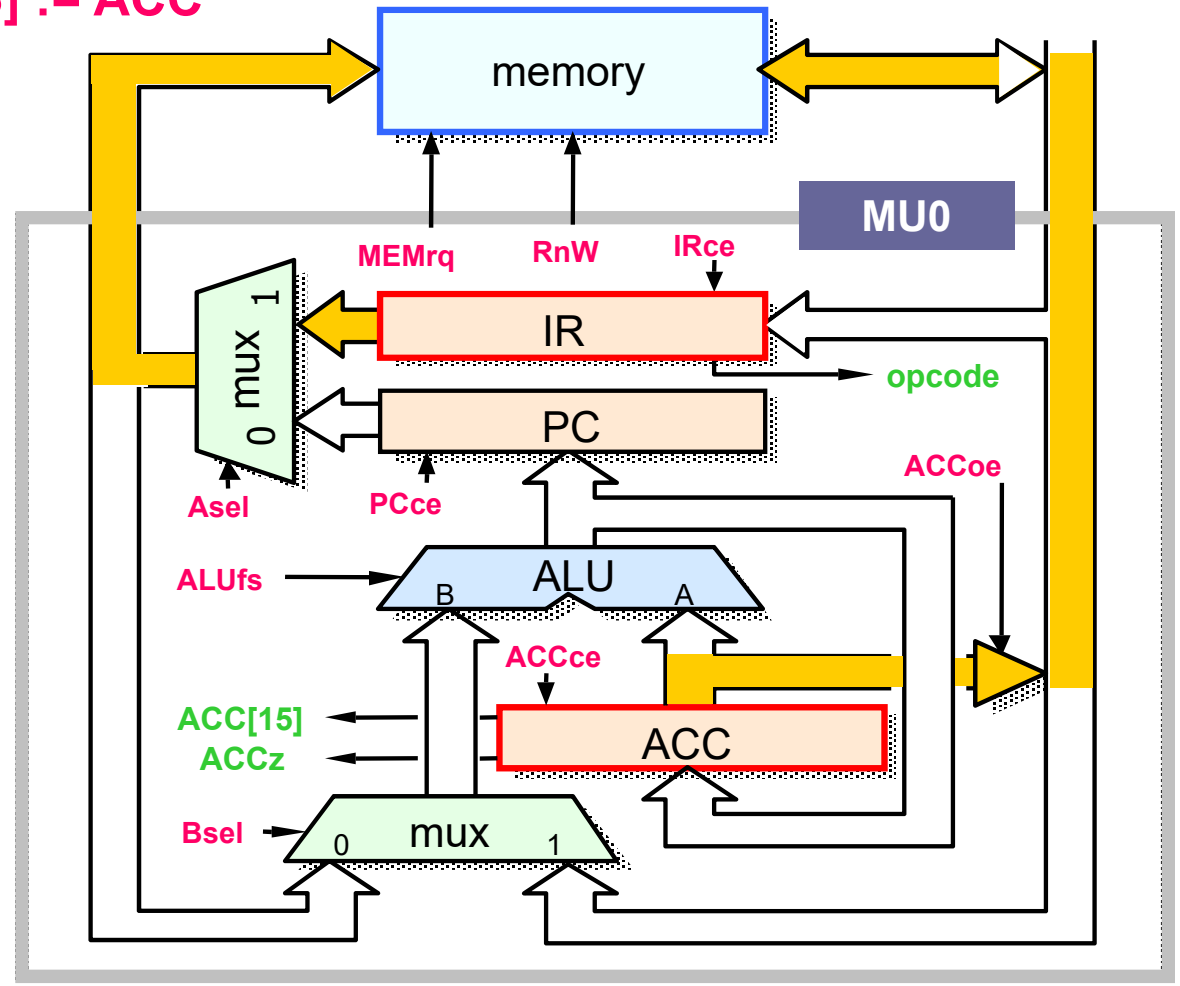


Inputs						Outputs									
Instruction	Opcode	Reset	Ex/ft	ACC15	ACCz	Asel	Bsel	ACCce	PCce	IRce	ACCoe	ALUfs	MEMrq	RnW	Ex/ft
LDA S	0000	0	0	x	x	1	1	1	0	0	0	= B	1	1	1
	0000	0	1 ft	x	x	0	0	0	1	1	0	B+1	1	1 R	0 Ex



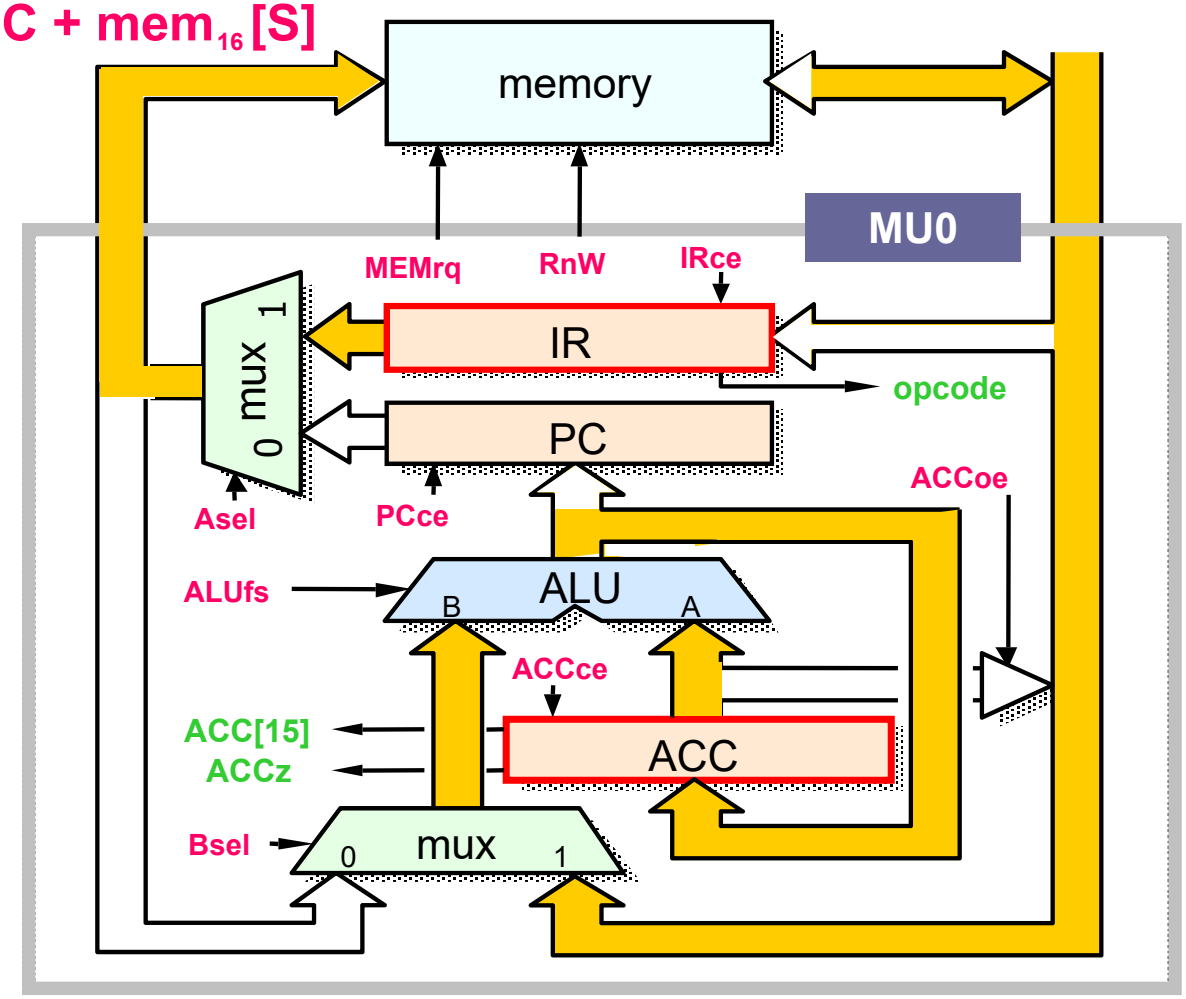
Inputs						Outputs									
Instruction	Opcode	Reset	Ex/ft	ACC15 ACCz		Asel	Bsel ACCce	PCce IRce	ACCoe		MEMrq RnW	Ex/ft ALUfs			
STO S	0001	0	0Ex	x	x	1	x	0	0	0	1	x	1	0W	1ft
	0001	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0

STO S $mem_{16}[S] := ACC$



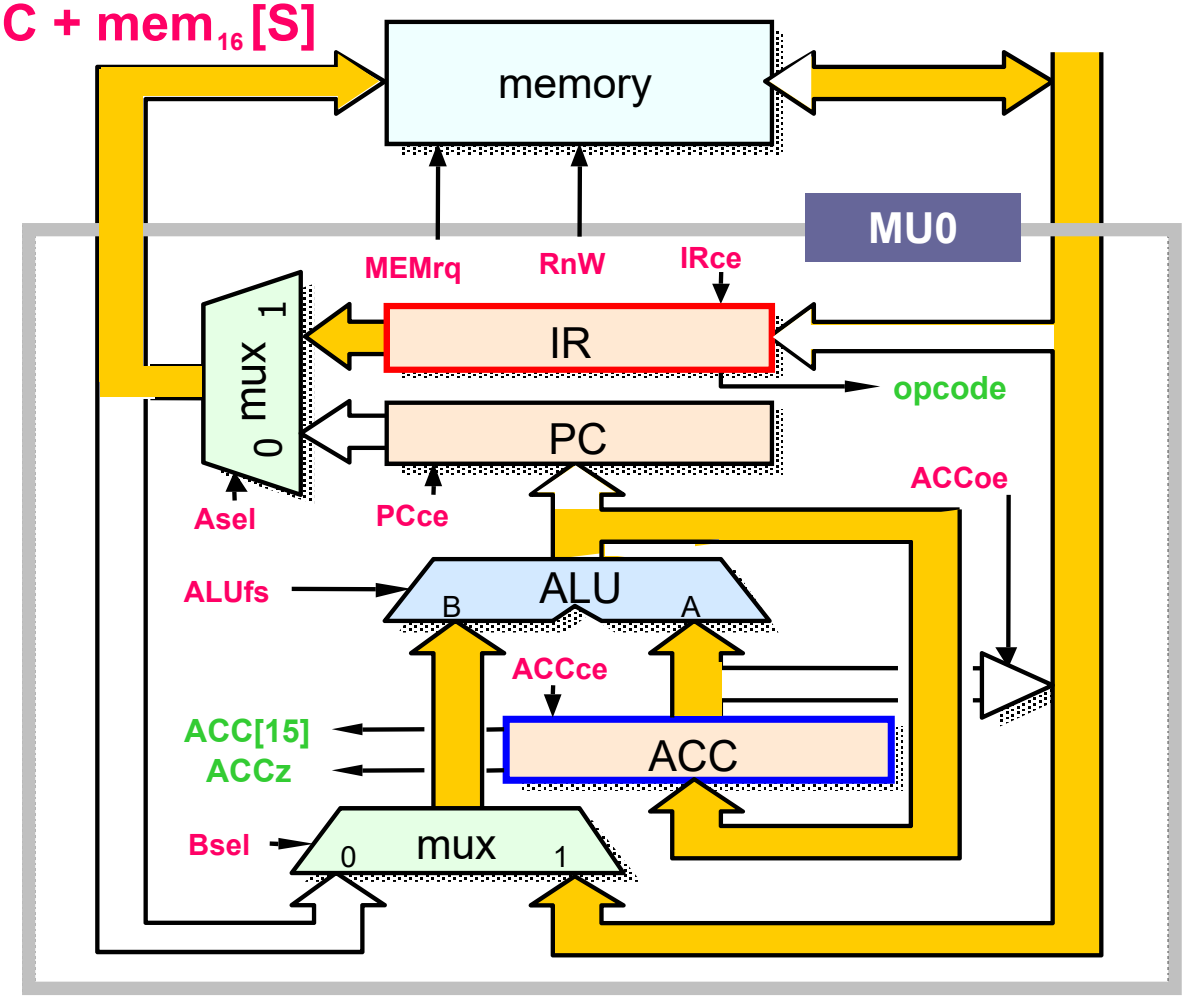
Inputs						Outputs									
Instruction	Opcode	Reset	Ex/ft	ACC15 ACCz		Asel	Bsel ACCce	PCce IRce	ACCoe	MEMrq	Ex/ft RnW	ALUfs			
ADD S	0010	0	0	Ex	x x	1	1	1	0	0	0	A+B	1	1	R ft
	0010	0	1		x x	0	0	0	1	1	0	B+1	1	1	0

ADD S ACC := ACC + mem_{16}[S]



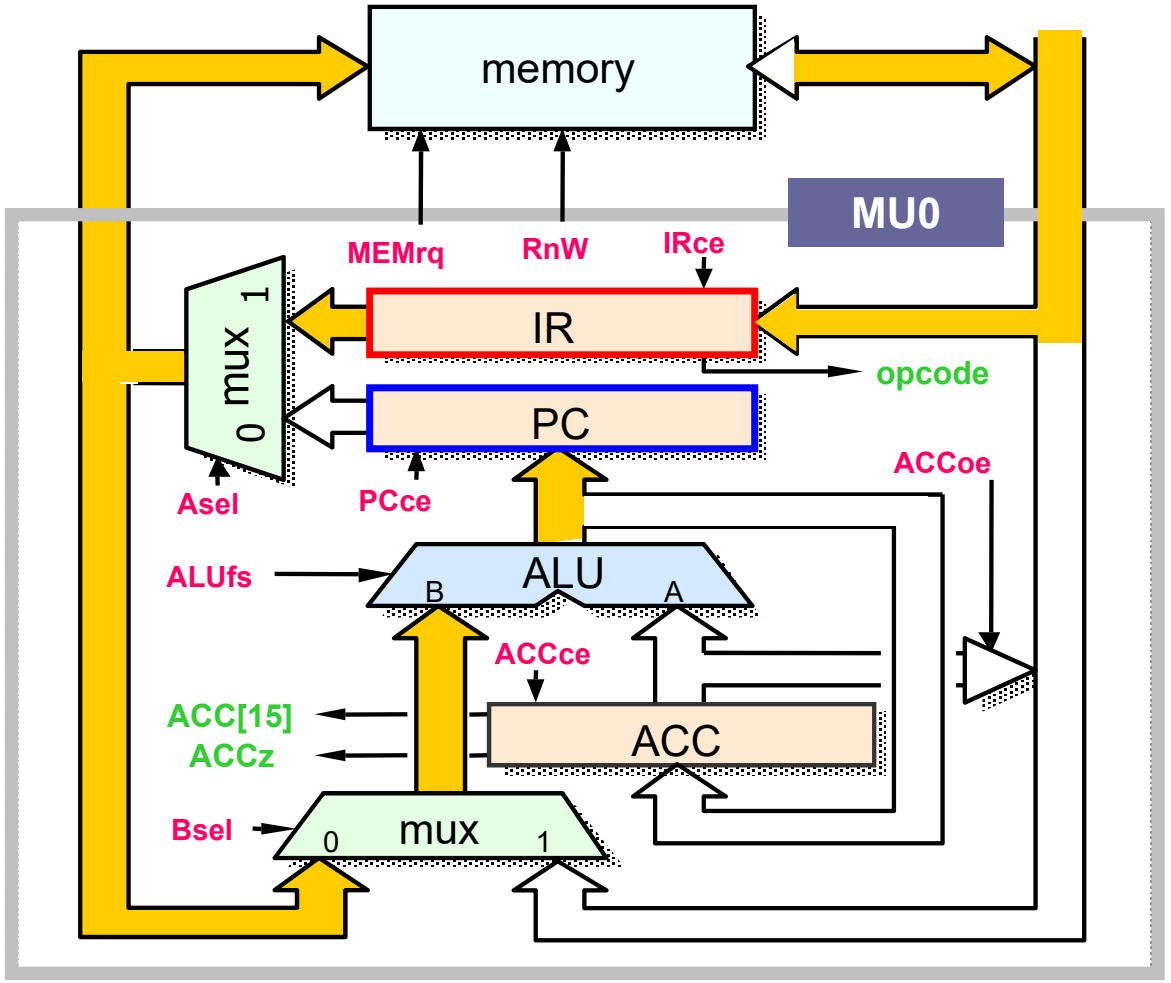
Inputs						Outputs									
Instruction	Opcode	Reset	Ex/ft	ACC15 ACCz		Asel	Bsel ACCce	PCce IRce	ACCoe	MEMrq	ALUfs	Ex/ft RnW			
ADD S	0010	0	0	Ex	x	x	1	1	1	0	0	0	A+B	1	1	R	1	ft
	0010	0	1		x	x	0	0	0	1	1	0	B+1	1	1		0	

ADD S ACC := ACC + mem$_{16}$[S]



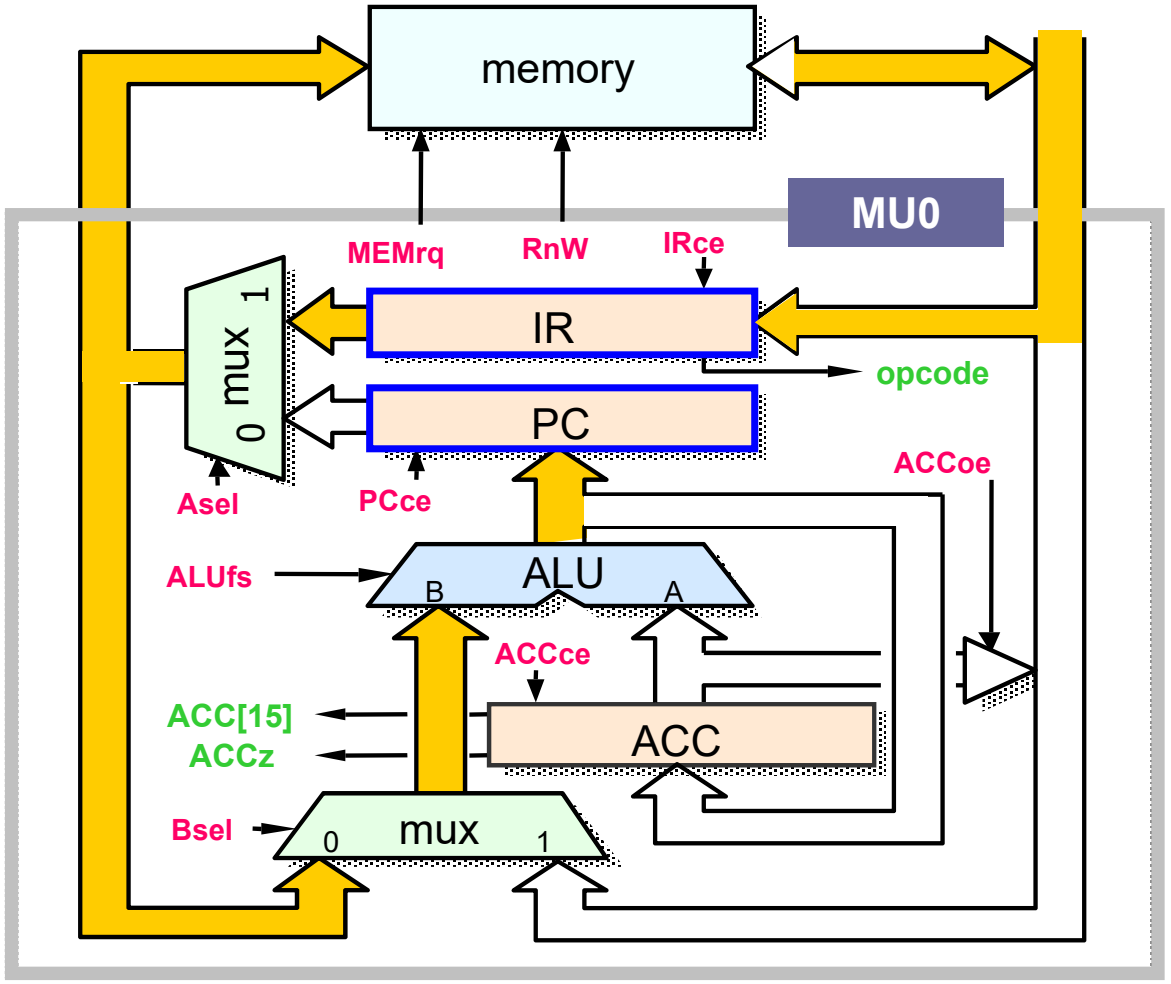
Inputs						Outputs									
Instruction	Opcode	Reset	Ex/ft	ACC15	ACCz	Asel	Bsel	ACCce	PCce	IRce	ACCoe	ALUfs	MEMrq	RnW	Ex/ft
JMP S	0100	0	x	x	x	1	0	0	1	1	0	B+1	1	1 R	0 Ex

JMP S PC := S



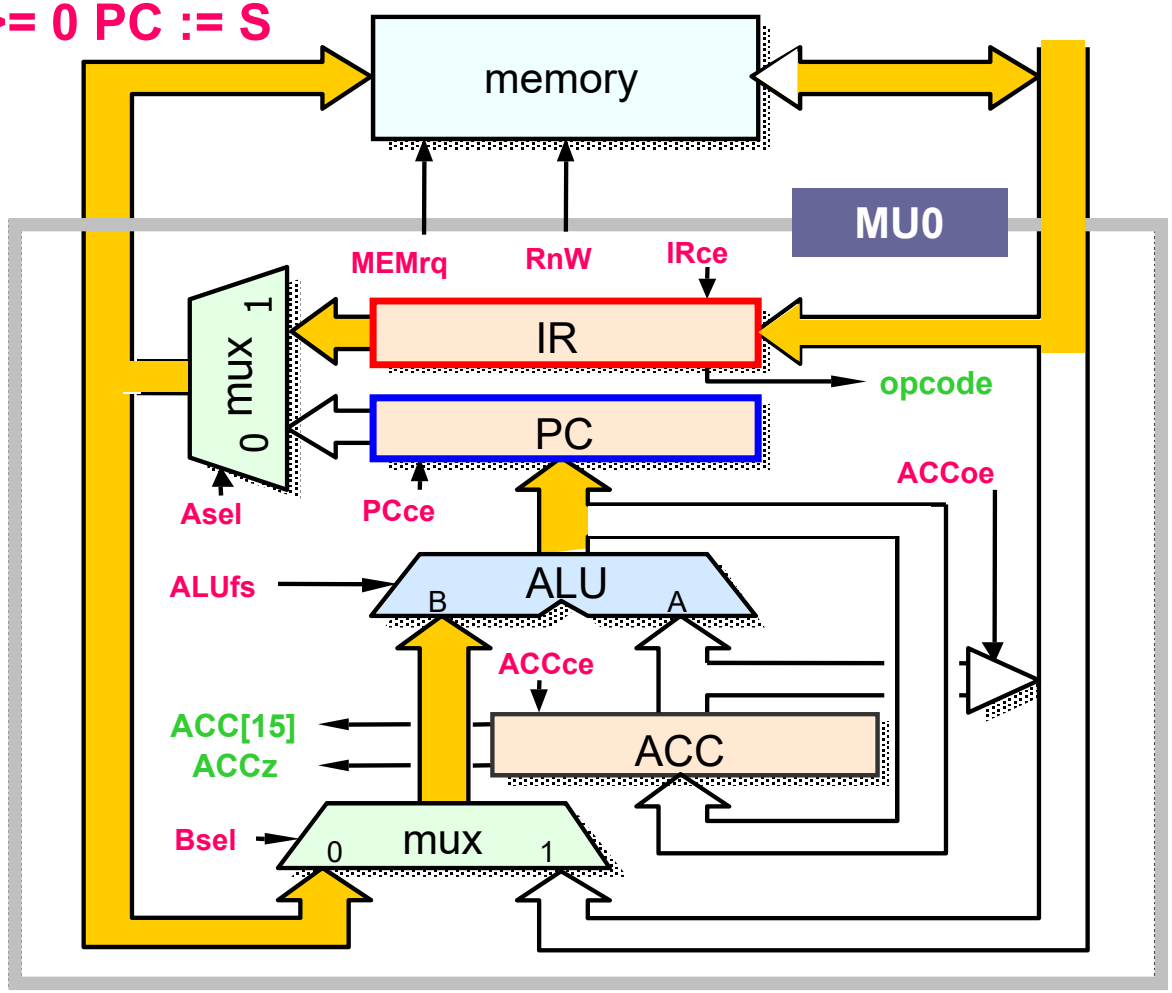
Inputs						Outputs									
Instruction	Opcode	Reset	Ex/ft	ACC15	ACCz	Asel	Bsel	ACCce	PCce	IRce	ACCoe	ALUfs	MEMrq	RnW	Ex/ft
JMP S	0100	0	x	x	x	1	0	0	1	1	0	B+1	1	1 R	0 Ex

JMP S PC := S



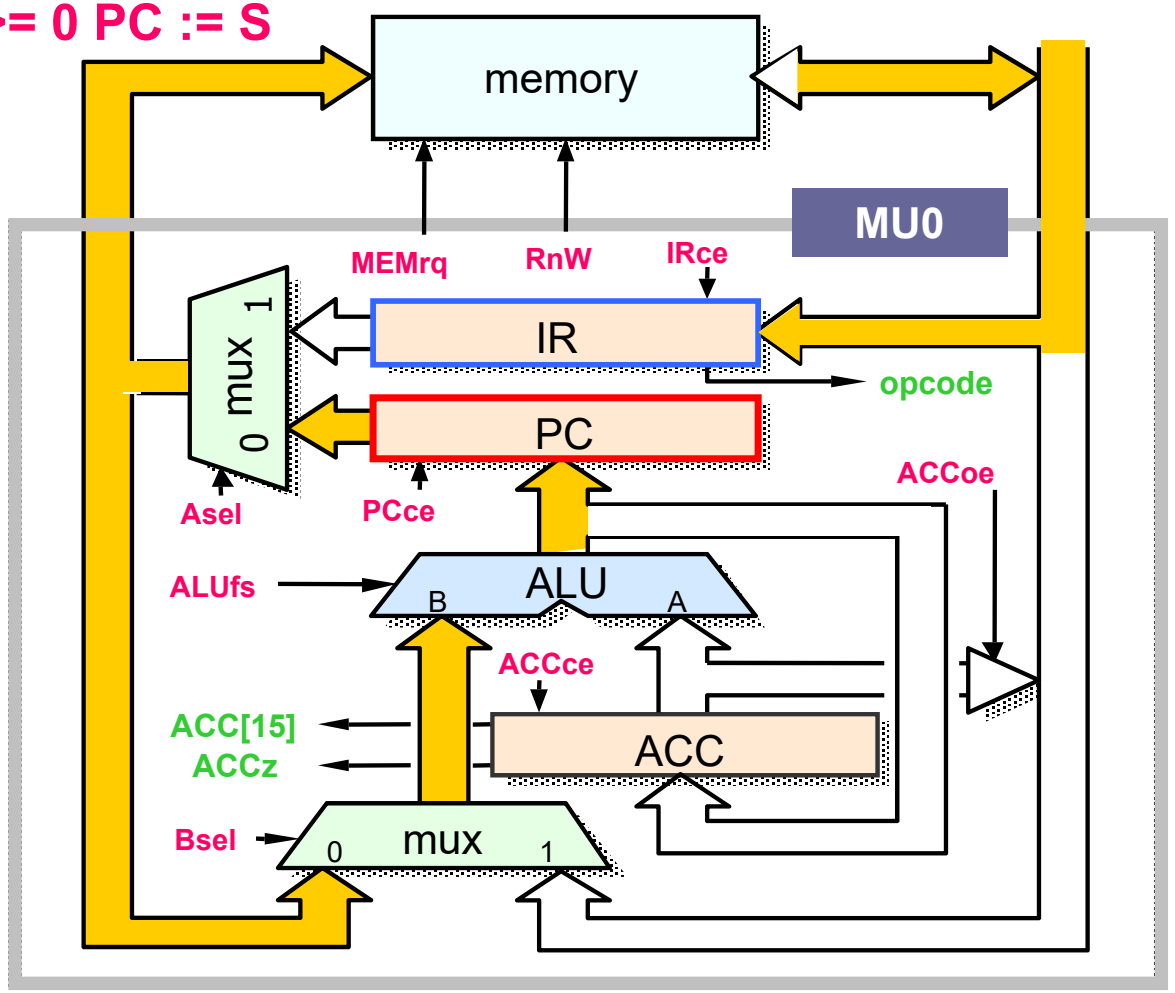
Inputs				Outputs									
Instruction	Opcode	Reset	ACCz	Asel	ACCce	IRce	ALUfs	Bsel	PCce	ACCoe	MEMrq	RnW	Ex/ft
JGE S	0101	0	x	x ≥ 0	1	0	0	1	1	0	B+1	1	1	R	0 Ex
	0101	0	x	x < 0	0	0	0	1	1	0	B+1	1	1		0

JGE S if ACC >= 0 PC := S

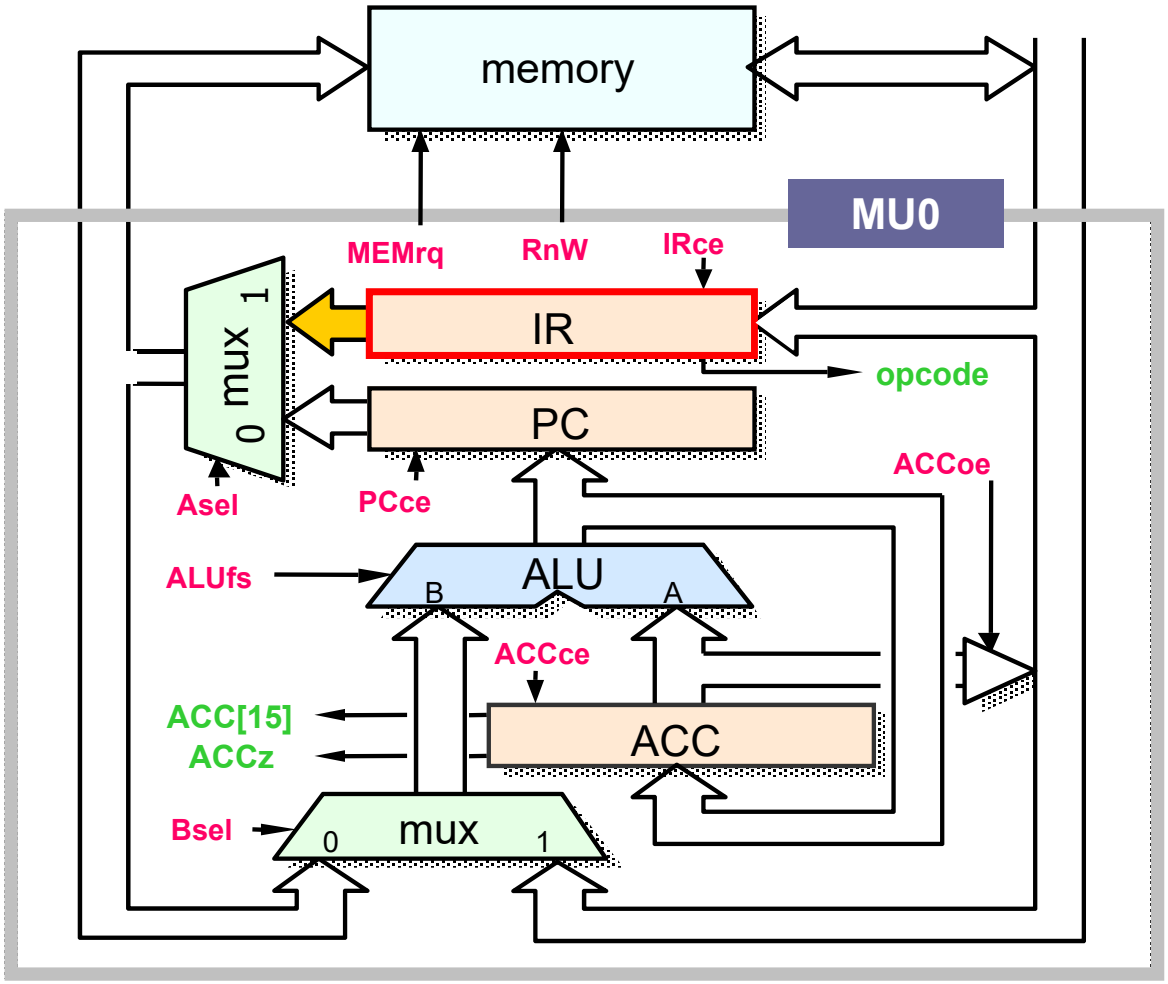


Inputs					Outputs									
Instruction	Opcode	Reset	Ex/ft	ACC15 ACCz	Asel	Bsel	ACCce	PCce	IRce	ACCoe	ALUfs	MEMrq	RnW	Ex/ft
JGE S	0101	0	x	x ≥ 0	1	0	0	1	1	0	B+1	1	1	0
	0101	0	x	x < 0	0	0	0	1	1	0	B+1	1	1 R	0 Ex

JGE S if ACC >= 0 PC := S



Inputs						Outputs									
Instruction	Opcode	Reset	Ex/ft	ACC15	ACCz	Asel	Bsel	ACCce	PCce	IRce	ACCoe	ALUfs	MEMrq	RnW	Ex/ft
STOP	0111	0	x	x	x	1	x	0	0	0	0	x	0	1	R 0 Ex



Execution cycles for MU0 programs

- Mem[0] LDA #20
- Mem[1] ADD #30
- Mem[2] STO #40
- Mem[3] JMP # 0

- Reset/IF (instruction fetch)
  - LDA #20
- T1 IE (instruction execution)
  - LDA #20
- T2: IF
  - ADD #30
- T3: IE
  - ADD #30
- T4: IF
  - STO #40
- T5: IE
  - STO #40
- T6: IF
  - JUMP#0
- T7: IE/IF
  - LDA #20

How to translate the following C code

- int i=4, j=5, k;
- k = i + j

How to support a simple for-loop application?

- for (i=0; i<10; i++) total = total + i;

The use of general register file

- Consider the MU0 implementation of the following function:
  - D=(A+B)(A+C)(B+C)
- MU0 assembly program
  - LDA #MEM[A]
  - ADD #MEM[B]
  - STO #MEM[X1]
  - LDA #MEM[A]
  - ADD #MEM[C]
  - STO #MEM[X2]
  - LDA #MEM[B]
  - ADD #MEM[C]
  - MUL #MEM[X2]
  - MUL #MEM[X1]

So, what's the problem?

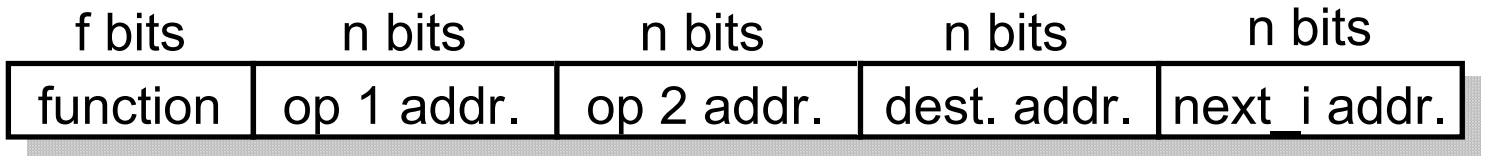
How to improve?

1.3 MU0 – a simple processor

- MU0 extensions
  - Extending the **address space**
    - **int sample[10000];**
  - Adding more **addressing modes**
    - **ADD #10 => ACC:=ACC+10;**
  - Allowing the PC to be saved in order to support a **subroutine** mechanism
    - Main(){
    - int a, b, c;
    - a=b+c;
    - sub();
    - a=b-c;
    - sub();}
  - Adding **more registers**
  - supporting **interrupts**, and so on …

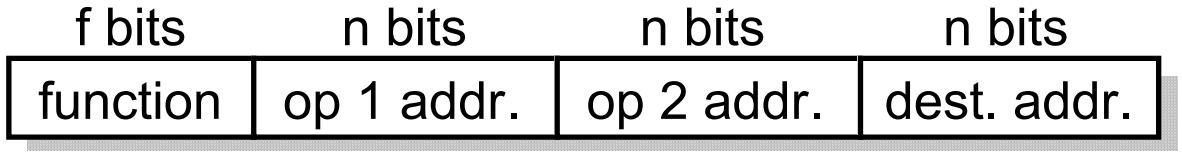
1.4 Instruction set design

- 4-address instructions

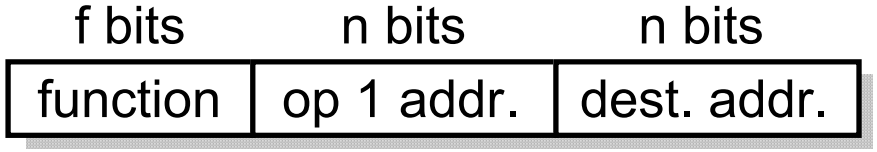


address of next instruction

- 3-address instructions



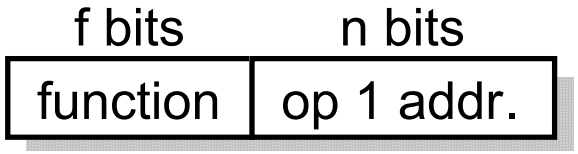
- 2-address instructions



source and destination register

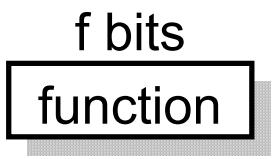
1.4 Instruction set design

- 1-address instructions



Accumulator ⇒ source and destination register

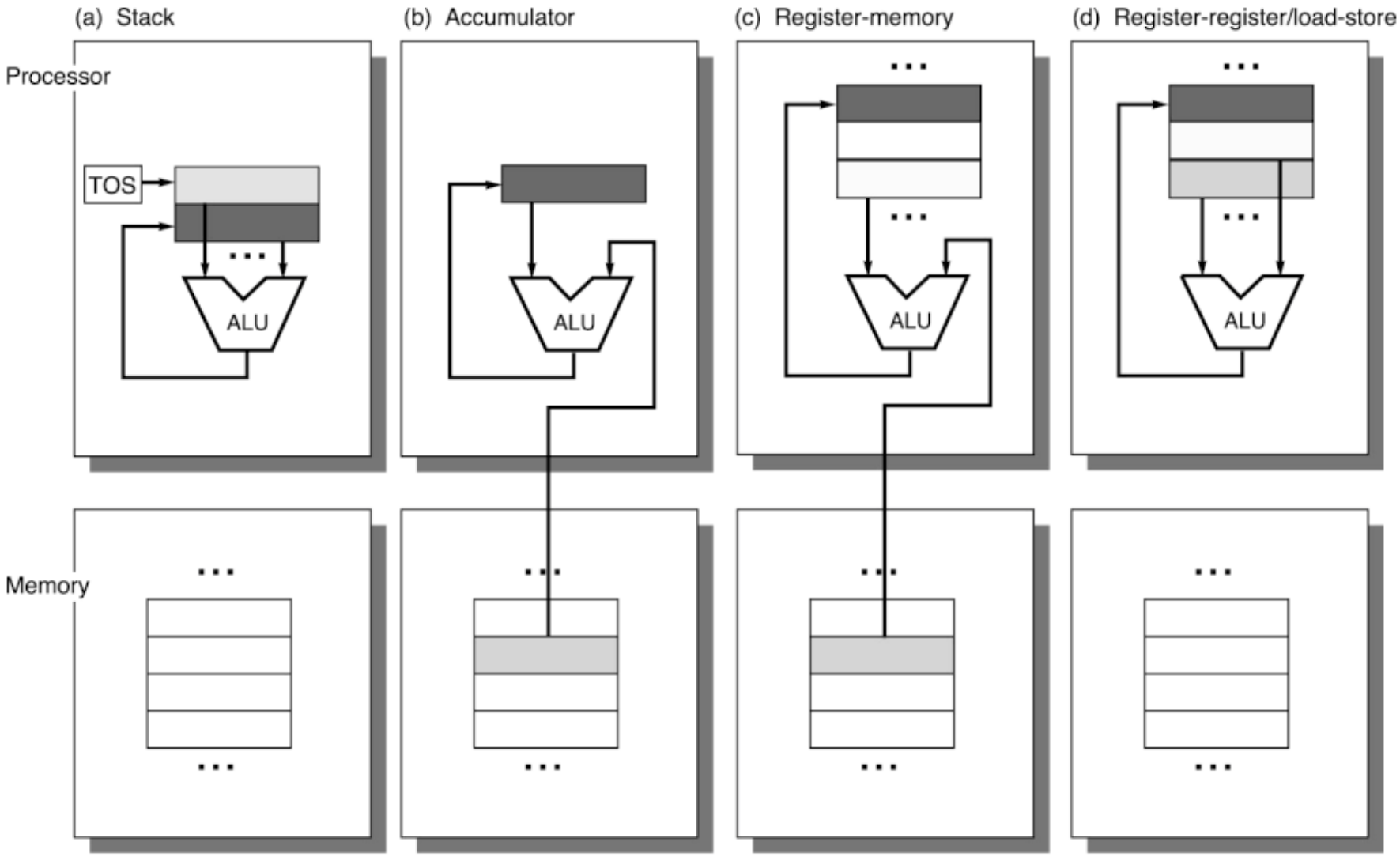
- 0-address instructions



stack

1.4 Instruction set design

- Examples of n-address use
  - 0-address ~ 3-address
  - The Thumb instruction set used for high code density on some ARM processors uses an architecture which is predominantly of the 2-address form
  - The standard ARM instruction set uses a 3-address architecture
- Instruction types
  - **Data processing instructions** such as add, subtract and multiply
  - **Data movement instructions** that copy data from one place in memory to another,or from memory to the processor's registers, and so on
  - **Control flow instruction** that switch execution from one part of the program to another, possibly depending on data values
  - **Special instructions** to control the processor's execution state



Operand locations for four instruction set architecture classes

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R3,R1,B	Load R2,B
Add	Store C	Store R3,C	Add R3,R1,R2
Pop C			Store R3,C

Fig. The code sequence for C = A + B for four classes of instruction sets

1.4 Instruction set design

- Addressing modes: how to specify the desired ***operands***
  - Immediate addressing
    - the desire value is encoded in the instruction.
      - ADD #6: ACC=ACC+6 (x=x+6)
  - Absolute addressing (direct addressing)
    - the address of the desired value in memory is encoded in the instruction.
      - ADD 0x01f0: ACC=ACC+mem[0x01f0] (x=x+y)
  - Indirect addressing
    - The instruction contains the address of memory location that contains the binary address of the desired value.
      - ADD, [0x1f0]: ACC=ACC+mem[mem[0x01f0]] (x=x+*y)
  - Register addressing
      - ADD r1: ACC=ACC+r1
  - Register indirect addressing
      - ADD [r1]: ACC=ACC+mem[r1]

1.4 Instruction set design

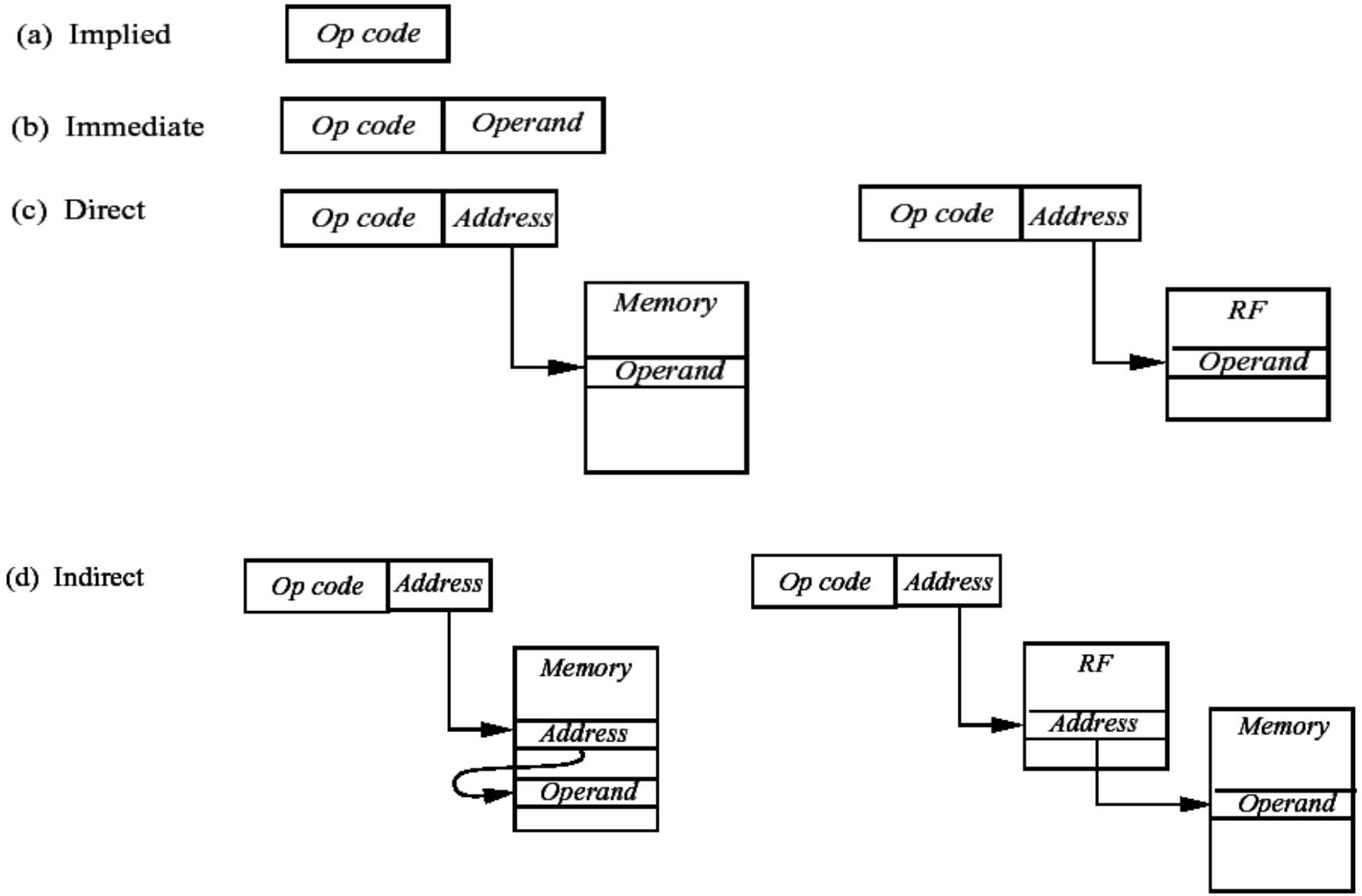
Addressing modes

- Base plus offset addressing
  - The instruction specifies a register as base and a binary offset to be added to the base to form the memory address
    - ADD r1, #6 : ACC=ACC+mem[r1+6] (x=x+a[6])
- Base plus index addressing
  - specify a base register and another register (the index)
    - ADD r1, r2 : ACC=ACC+mem[r1+r2] (x=x+a[i])
- Base plus scaled index addressing
  - as above, but the index is multiplied by a constant (usually the size of the data item, and usually a power of two)
    - ADD r1, r2, #3: ACC=ACC+mem[r1+r2.2<<3]
- Stack addressing
    - ADD: ACC=ACC+mem[SP]

9.2 Addressing modes

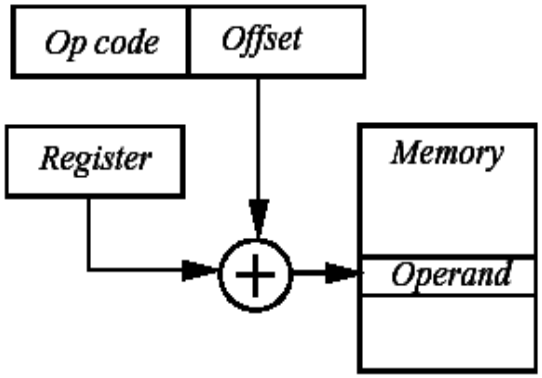
- The most frequently used addressing modes
  - *Immediate addressing mode*
    - the operand (constant) is specified in the address field
  - *Direct addressing mode*
    - the address field specifies the location (in memory or register file) of an operand or the result
  - *Indirect addressing mode*
    - the address field specifies the location of the address of an operand or the result
  - *Relative addressing mode*
    - the content of the address field (offset) is added to the content of a specified register (program counter or a register in register file)
  - *Indexed addressing mode*
    - the address field specifies a starting address (base), the index of a particular data is specified in a register
    - access data is stored in arrays, stacks, and queues

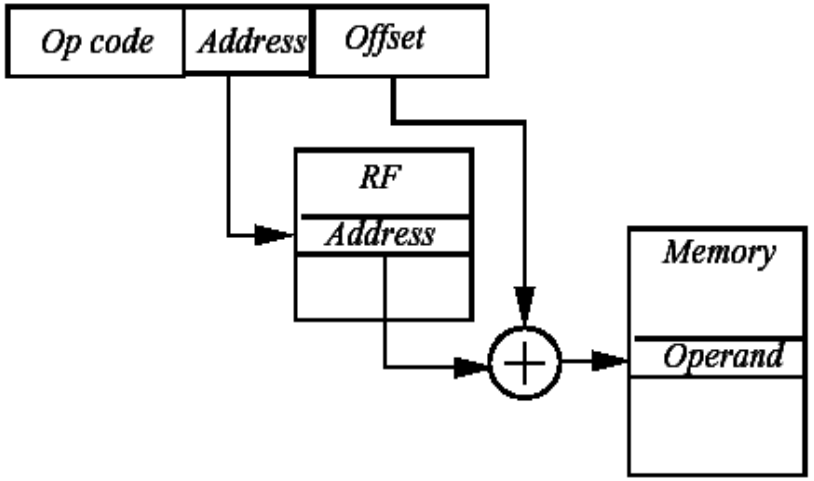
Addressing modes



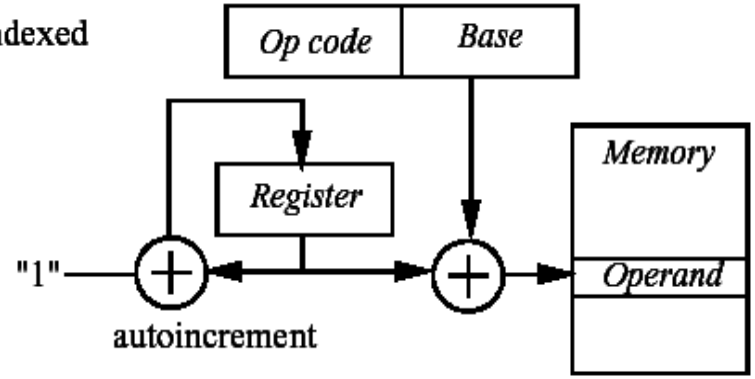
Addressing modes

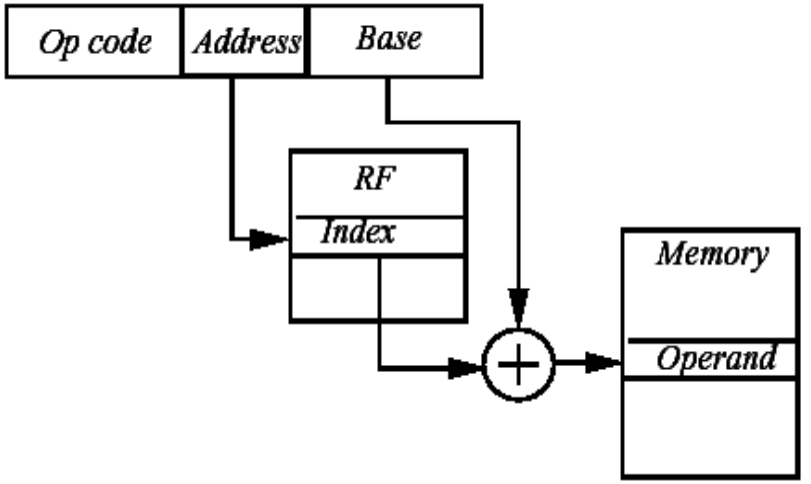
(e) Relative





(f) Indexed





1.4 Instruction set design

- Control flow instructions
  - Control flow instructions
    - where the program must deviate from the default (normally sequential) instruction sequence, a control flow instruction is used to modify the program counter (PC) explicitly
  - The simplest such instructions are usually called 'branches' or 'jumps'
  - The maximum range of the branch is determined by the number of bits allocated to the displacement in the binary format
- Conditional branches
  - Some processors allow the values in the general registers to control whether or not a branch is taken through instructions such as:
    - Branch if a particular register is zero (or not zero, or negative, and so on)
    - Branch if two specified registers are equal (or not equal)

1.4 Instruction set design

- Subroutine calls
  - Since the subprogram may be called from many different places, a record of the calling address must be kept
    - The calling routine could compute a suitable return address and put it in a standard memory location for use by the subprogram as a return address
    - The return address could be pushed onto a stack
    - The return address could be placed in a register
  - Most architectures include specific instructions to do the subprogram calls.
- Subprogram return
  - Return instruction
    - move the return address from wherever it was stored (memory, stack, or register) back into the PC

1.4 Instruction set design

- System calls
  - This is a branch to an **operating system routine**, often associated with a change in the privilege level of the execute program
  - Some functions in the processor, possibly including all the input and output peripherals, are protected from access by user code
    - User program that needs to access these functions must make a system call
  - A well-designed processor will ensure that it is possible to write a multi-user operating system where one user's program is protected from assaults from other users
- Exceptions
  - The change in the flow of control is a consequence of some **unexpected** side-effect of the program
    - Access a memory location may fail because a fault is detected in the memory subsystem

1.5 Processor design trade-offs

- The art of processor design is to define an instruction set that supports the functions that are useful to the programmer while allowing an efficient implementation.
- The semantic gap between a high-level language construct and a machine instruction is bridged by a compiler.
- The processor designer should define the instruction set to be a good compiler target rather than something that the programmer will use directly to solve the problem by hand.

1.5 Processor design trade-offs

- Complex Instruction Set Computer
- Reduced Instruction Set Computer
- What processors do
  - It is a common misconception that computers spend their time computing, that is, carrying out arithmetic operations on user data
  - Although they do a fair mount of arithmetic, most of this is with addresses in order to locate the relevant data items and program routines
  - At the instruction set level, it is possible to measure the frequency of use of the various different instructions
  - Table1.3 were gathered running a print preview program on an ARM instruction emulator

1.5 Processor design trade-offs

Instruction type	Dynamic usage
Data movement	43%
Control flow	23%
Arithmetic operations	15%
Comparisons	13%
Logical operations	5%
Other	1%

Table 1.3 Typical dynamic instruction usage

1.5 Processor design trade-offs

- Making processors go faster
  - **Pipelining**: the most important one
  - Use of a **cache memory**
  - Super-scalar instruction execution: is very complex, has not been used on ARM processors
- Pipelines
  - Fetch: fetch the instruction from memory
  - Dec: decode it to see what sort of instruction it is
  - Reg: access any operands required from the register bank
  - ALU: combine the operands to form the result or a memory address
  - Men: access memory for a data operand, if necessary
  - Res: write the result back to the register bank

1.5 Processor design trade-offs

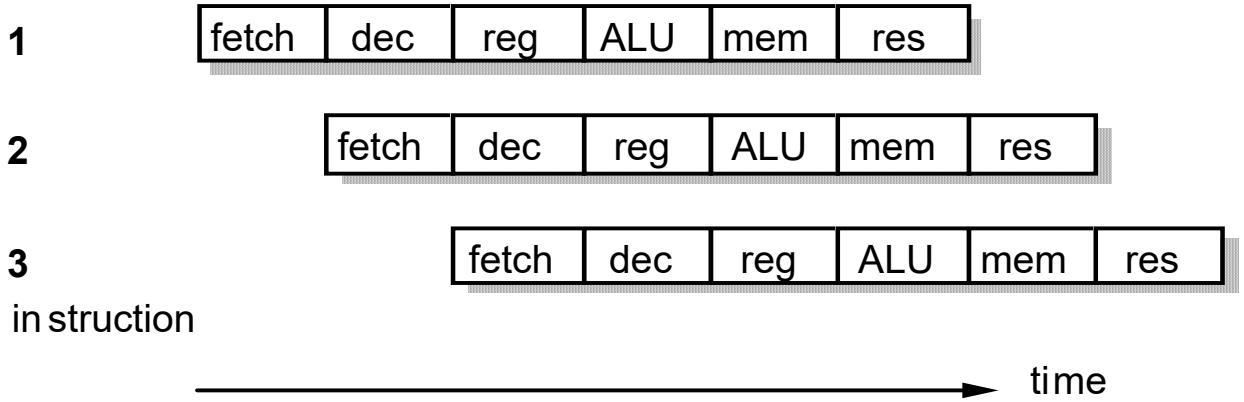


Fig. 1.13 Pipelined instruction execution

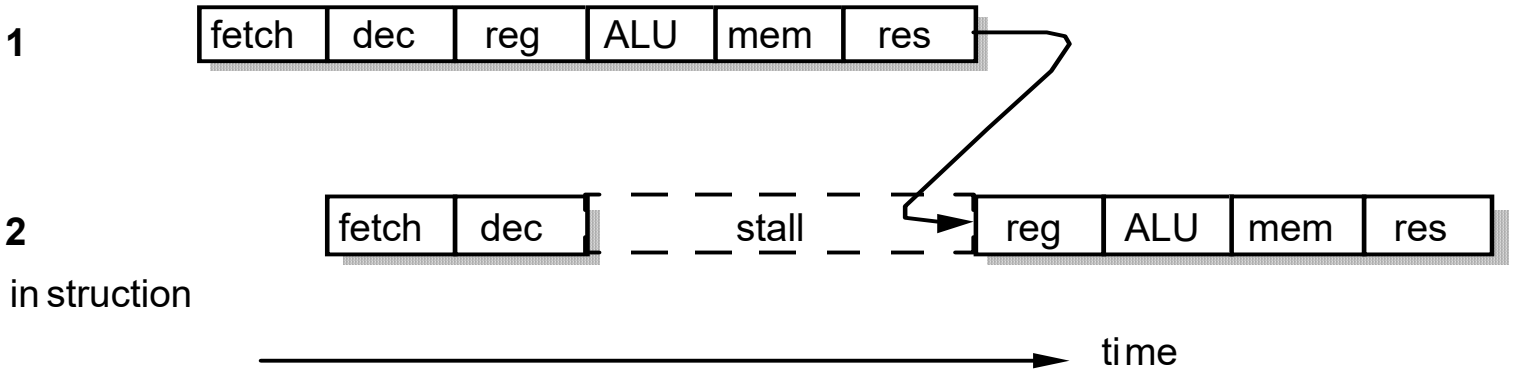


Fig. 1.14 Read-after-write pipeline hazard

1.5 Processor design trade-offs

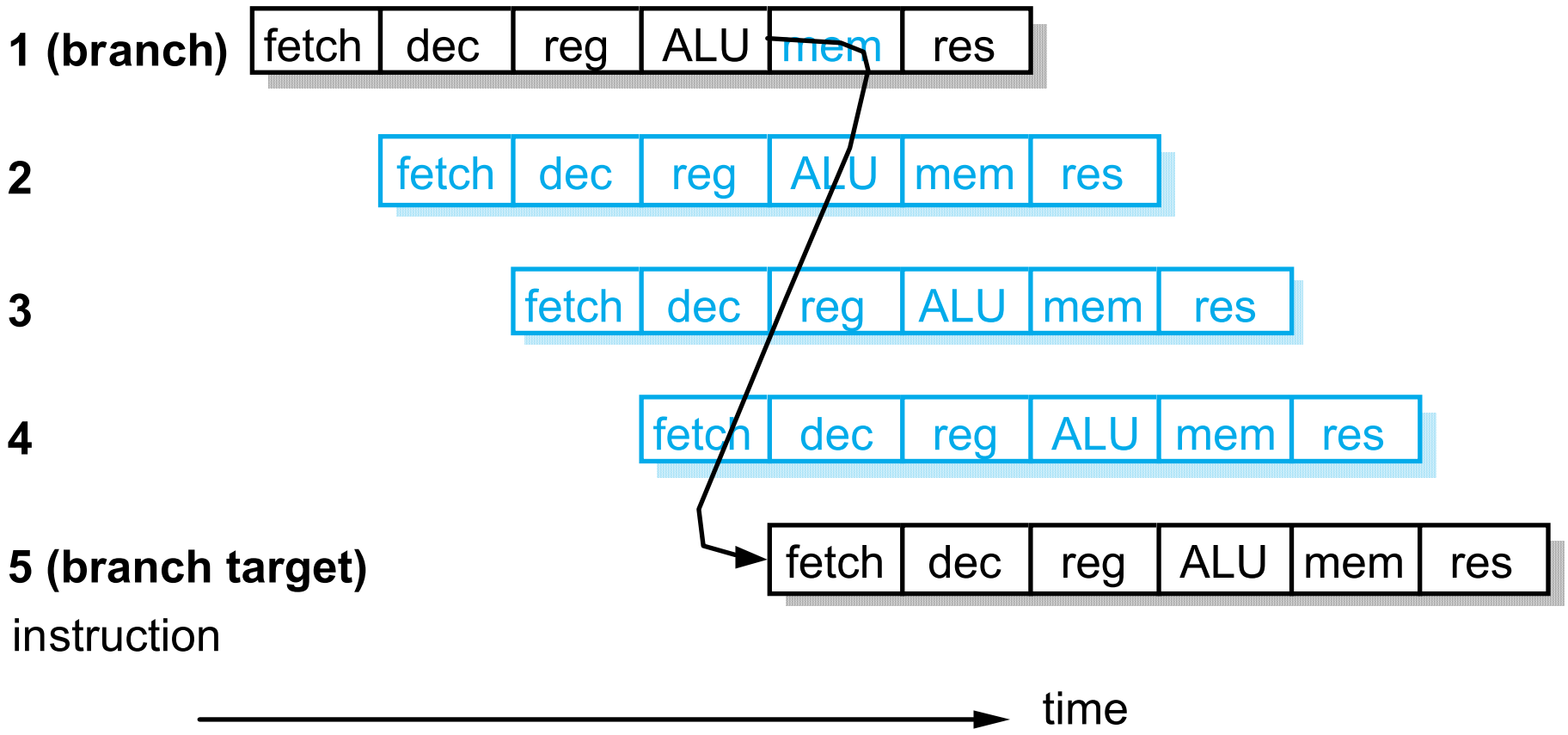


Fig. 1.15 Pipelined branch behaviour

1.5 Processor design trade-offs

- Pipeline efficiency
  - There are techniques which reduce thee impact of these pipeline problems
  - The **deeper** the pipeline, the **worst** the problems get
  - For reasonably simple processors, there are significant benefits introducing pipelines from three to five stages long
  - Processors with very complex instructions where every instruction behaves differently from the next are hard to pipeline
  - In 1980 the complex instruction set microprocessor of the day was not pipelined